About This Book

OS/2 Warp 4 consists of various products including OS/2 Warp Version 4, BonusPak, IBM's OS/2 implementation of Sun Microsystems, Inc.'s Java technology, VoiceType for OS/2 Warp, and networking products.

This book provides programmers with guide and reference information for collecting and managing problem determination data using OS/2 Warp Version 4.

This book first provides conceptual and introductory information about OS/2 Warp Version 4 First Failure Support Technology (FFST). This sophisticated and powerful tool requires thoughtful planning and code instrumentation. These topics present planning, setup, and examples that are used for collecting and managing problem determination data.

Separate sections are devoted to understanding the aids that are provided for tracking, collecting, storing, and formatting problem determination data, such as traces, dumps, and error logs.

After reading this book you should understand the benefits of FFST and know how to instrument your code. This allows you to take advantage of the technology and tools that are associated with problem determination data. You should also know how to use the APIs, trace, error logging, and dump functions. These functions retrieve, format, and analyze problem determination data.

This book assumes that you have the IBM Developers Toolkit for OS/2 Warp Version 4 installed and you are developing application programs. In the chapters of this book refer to the IBM Developers Toolkit simply as the Toolkit.

Related Publications

The following publications contain additional information about the OS/2 Warp Version 4 product:

- *OS/2 Warp Version 4 CP Reference* in the Toolkit
- *OS/2 Warp Version 4 Command Reference* in the Toolkit
- *Trace* document in the Serviceability and Diagnostic Aids folder if the folder is installed
- *Trace Customizer* in the Toolkit
- *DMI Programmer's Guide* in the Toolkit
- *SystemView Agent Client Guide* in the Tasks folder

Introduction to Collecting and Managing Problem Determination Data

Ideally, programs run forever, error-free, and within their performance targets. However, code failures do occur. When this happens, programmers need problem analysis tools and techniques that help find problems as quickly as possible.

OS/2 Warp Version 4 provides the architecture, tools, and support to help you collect and manage problem determination data. The goal is to get your code back on track (that is, **service the code**) as quickly as possible.

This chapter introduces the problem analysis elements and techniques that are available to you. The remainder of the book provides details about these elements.

An Approach to Collecting and Managing Problem Determination Data

Although there is no right or wrong way to repair code problems, this book presents a three-step approach to collecting and managing problem determination data.

Step 1 - Check the Error Log and Data Areas

The first places you should check when code failures occur are the Error Log and associated data areas. OS/2 Warp has a technology called FFST (pronounced "fist"), which stands for First Failure Support Technology. A failure in code that uses FFST technology collects useful problem analysis information that is stored in the error log and in data areas. This can happen without user or programmer intervention. Error log data provides information such as the date and time an error occurred. This data also identifies the module in which the error occurred, the severity, and description of the error. You can use data areas to store user-specified information (data structures, for example) when code fails. The system saves the data area with the error log entry. Many code problems can be pinpointed by analyzing what the error log and data areas tell you. A utility called SYSLOG controls error logging and works with error log contents.

How does a programmer capture problem determination data? FFST is the answer. FFST is a technique of capturing problem determination data at the time of a code failure. You use FFST by placing calls to the FFSTProbe API in strategic places in your code. Each time you call FFSTProbe for a problem, the system writes an entry to the error log. The system also writes entries to data areas if you have coded the FFSTProbe API to do so. The data in the error log entry includes information that identifies the product that encountered the problem. The system stores product information in a database that conforms to the Desktop Management Interface (DMI) standards.

FFSTProbe is a powerful function and requires some planning and setup. You **instrument** your code by placing calls to FFSTProbe at specific points in your code, along with your instructions for the function. Guide to Instrumenting Your Code, provides an introduction to using this API and steps for planning and using it.

You may decide later that you want to override the calls to FFSTProbe. A Probe Control Table, with a graphical utility, provides this function. Entries in the Probe Control Table override the coded probe functions. This dynamic override capability is very useful because it allows you to change what the probes do without having to recode and recompile your programs.

Guide to Instrumenting Your Code, provides detailed information about this first phase of problem analysis. It explains the data collected and provides guidance for planning and instrumenting your code. Capturing and Saving Failure-Related Information through Dumps, provides more details about how to use the PM Dump Facility dump formatter to view the FFST dump. Viewing and Analyzing Error Log Entries, provides more details about the error log table, what the error log table contains, and how you can work with the log.

Step 2 - Use Trace Facilities

Step 1 requires no user or programmer intervention with failing code. Step 2 involves setting trace points and using trace data, and it does require intervention. The following functions are available in OS/2 Warp Version 4 to help you use trace effectively:

- A function to insert trace points in your code
- A command that describes the trace file to be used
- A command that turns trace on and off
- A utility called Trace Customization that lets you format entries in the trace file
- A utility called Trace Formatter that displays the contents of the trace file

Traces allow you to see and follow the course of events in code that lead to a failure. You can use trace data:

- To understand the order or determine the operating path of the code
- To understand parameter data changes during processing
- To examine inputs to functions and outputs from functions
- To examine resulting return codes
- To save intermediate data values

Analyzing Performance and Debugging Problems Using Trace, contains more information about the Trace facilities.

Step 3 - System Dump

If steps 1 and 2 do not help you determine the cause of a code failure, step 3, system dump, is the recommended final step. This is the primary tool used by service personnel to solve system code problems and application code problems. Use a debugger for application problems. OS/2 Warp Version 4 has the technology to initiate a system dump when code fails. The dump information that is stored on disk provides information about the reasons for the code failure. Use this step as a final step because your system will restart after it stores the dump.

This third step of problem analysis has three phases:

1. Configure: involves adding the TRAPDUMP statement to the CONFIG.SYS file, selecting the *Enable System Dump* choice on the Probe Control Table, and allocating disk space by using a disk partition for storing the system dump. System dumps can be stored on diskettes but the number of diskettes required will depend on the amount of main storage memory your system has.

2. Trigger: the way you start a system dump:

    - by a keyboard sequence
    - by calling the DosForceSystemDump API
    - by using the Probe Control Table (PCT) to override the values that are specified in a call to FFSTProbe
    - when an unhandled trap occurs.

3. Format and view: the step where you look at the system dump data using the PM Dump Facility dump formatter.

Refer to Capturing and Saving Failure-Related Information through Dumps for more information about System Dumps and the PM Dump Facility.

Summary

The information in this book describes First Failure Support Technology and the supporting trace and dump facilities that are used during problem analysis. Note that trace utilities are not part of FFST. The FFST technology provides the tools (functions, commands, and graphical interface utilities) to help you instrument and service your code to take advantage of this technology.

Summary of Functions and Interfaces provides a summary of the functions, commands, utilities, and interfaces that comprise FFST.

Although problem determination can be done, whether or not your code is instrumented, any time taken to instrument code is well spent. By instrumenting your code, you will be able to take full advantage of the FFST, trace, and dump tools if code problems occur.

Guide to Instrumenting Your Code

Code instrumentation improves problem analysis. Instrumented components of OS/2 Warp Version 4 use First Failure Support Technology (FFST) and trace. This chapter defines the required steps for instrumentation, and things you should consider before you instrument your code. This chapter also tells you what to expect when you use the FFSTProbe API and trace utility.

Introduction to FFST Instrumentation

FFST is a programming concept that uses a set of software tools and services to capture error information at the time of a code failure. You view the error information using system error log or PM Dump Facility dump formatter to determine the cause of the problem. You capture error information by placing a call to the FFSTProbe API in your code. You **instrument** your code by calling FFSTProbe and specifying which data to collect.

When your properly instrumented code encounters an unsuspected or unrecoverable error, the code immediately calls the FFSTProbe API to capture failure related information. Your code specifies the parameters to capture data when calling the FFSTProbe function. The system creates an error log entry each time your code calls the FFSTProbe function. The log entry will contain the information your code specifies in the call to FFSTProbe. After the call, the system returns control to your code unless the system triggered a system dump. System dumps automatically restart the system. Additional error information can be collected by using a Probe Control Table (PCT) entry. System dumps are triggered by using PCT entries. The captured information that is contained in the error log entry can include event trace data, program error information, or user-defined data.

Therefore, FFST consists of a collection of functions, commands, and utilities within the Problem Determination Tools folder. Use the utilities to do the following:

- collect problem determination data
- define the types of data collected
- specify where to store the collected error data

- override parameters on calls to the FFSTProbe function.

Summary of Functions and Interfaces, provides an overview of the interfaces to FFST. Problem Determination APIs, provides descriptions of the API functions.

This chapter provides the information you need to instrument your code. It may be helpful to have the *OS/2 Warp Version 4 Tools Reference* document available for reference while using this book. The associated references are available on the Toolkit CD ROM.

Benefits of Instrumenting for FFST

Instrumentation is key to providing adequate code serviceability. If problems occur, instrumented code allows you or service personnel to take full advantage of the FFST technology in OS/2 Warp Version 4. The system records problem determination data with no user or additional programmer intervention. Instrumentation decreases the need for reproducing user failures. System dumps and process dumps however do require intervention and problem reproduction.

The captured information that is recorded in the error log is essential to problem solving. An error log entry contains information that indicates the failing product, and the time the error occurred. By analyzing the captured information, you can determine the failing components, diagnose the causes of the error, and correct the problems.

Overview of FFSTProbe API

The FFSTProbe API is the key to problem analysis by signalling that your code has encountered a problem. FFSTProbe captures the requested data, and stores the data in the error log for use in problem analysis.

FFSTProbe Parameters

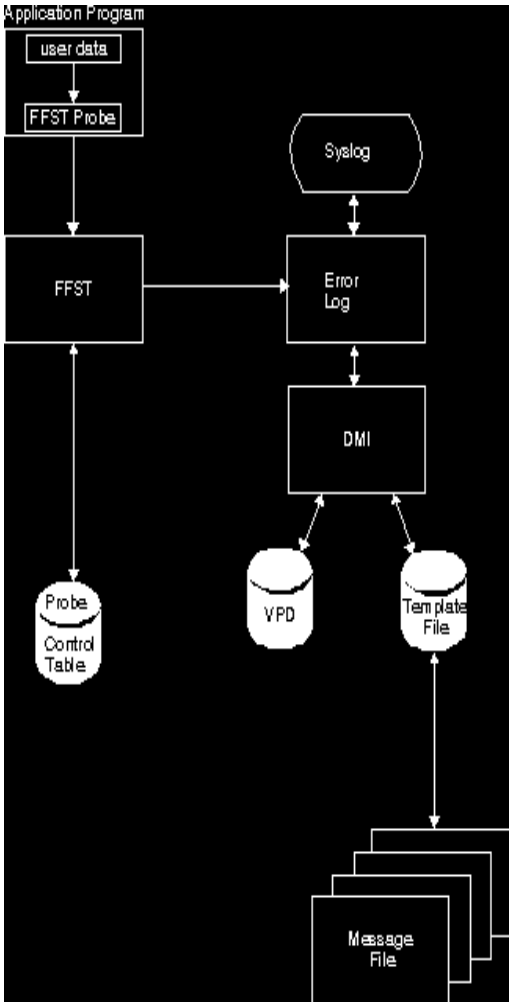
The FFSTProbe API parameters identify the product that reported the problem. The parameters specify which data to collect for the problem. You can use the parameters to specify the following:

- the severity of the call
- the associated error message data
- the name of the formatting template that is used to display the error information
- any system process information
- any specific user data that you want collected.

The system stores module name and time stamp automatically. The FFSTProbe API can also initiate a system dump to capture data that resides in the main memory of the system. Refer to Problem Determination APIs for the FFSTProbe API and its parameters.

FFST Flow

FFST Flow



The sequence of events that are shown in FFST Flow shows how FFST logs errors and captures data when your code calls FFSTProbe.

After you develop and install your code on the system, the application program box that is shown in the diagram above signifies your code. The FFSTProbe API is called when your code discovers a problem. If you specify to capture user data in the call to FFSTProbe, the system captures the data with the other error-related information.

Your code calls FFSTProbe to gather the following product information:

- dump information
- error message information
- other error-related data

The system records the data in the error log entry. If your code has entries in the Probe Control Table, FFST uses the entry values instead of the FFSTProbe parameters that are used in the calls. The system records the data in the error log entry. FFST uses the configuration values to create the error log entry.

After FFST gathers the error-related information, it stores the data as an error log entry. The system stores FFST dump information in a file named FFxxxxxx.DMP, where xxxxxx signifies a six-digit identifier. If a trace snapshot is requested, a file named FFxxxxxx.TRC will be created. If a process dump is requested, a file named FFxxxxxx.PRC will be created. The error log information contains the name of the FFST dump file along with the trace file if applicable.

Use the SYSLOG utility to view the error log information. SYSLOG uses message files and template files to format and display error log records. Use SYSLOG to control the following log functions:

- specify which error log file to use
- suspend or resume error logging
- change the size of the error log.

Steps for Instrumenting for FFST

The steps for instrumenting your code are as follows:

1. Planning for Instrumenting Your Code
2. Code the FFSTProbe API.
3. Compile the code.
4. Create the error record template file
5. Create message files.
6. Create DMI MIF files.

The remainder of the information in this chapter provides information about each step.

Planning for Instrumenting Your Code

There are several things to consider before you begin putting calls to the FFSTProbe API in your code. This section describes the following considerations and steps:

1. Define and ensure existence of Vital Product Data (VPD). VPD is the description of your code to the system. The system uses VPD to identify the product that is reporting a problem.
2. Decide how and where you should code your calls to the FFSTProbe function.
3. Decide what data you want the function to collect for code failures.

Defining Vital Product Data (VPD)

The DMI facility provides a standard way to register the hardware and software on the system. This allows both system software and system-based software (for example, application programs or device drivers) to register with the system. This information is called Vital Product Data (VPD). The system uses VPD to identify the source of error log entries. Various system management applications require access to the VPD information. When a product component uses FFSTProbe to log an error, the error logging function automatically includes the VPD information in the error record.

The VPD information allows systems management applications to assume a base level of VPD for all conforming products on a system. The VPD information for software products differs from the VPD information for hardware products. You can define additional specialized VPD information for your product.

Both your component's install object (that the feature installer uses to install your product) and the FFSTProbe parameter information must have identical information. This enables DMI to provide the template file that is specified on the call to FFSTProbe. Recommendations for these values are:

- Vendor - a description of the organization or company that developed the product that is reporting an error (example: IBM).
- Tag - a unique description of the product that is reporting an error (example: FFSTProbe SAMPLE).
- Revision - optional description of the development organization's revision level, change level, or version of the product that is reporting an error. (example: 1.0.1c). If a component within a product is reporting the error, the revision may not correspond to the revision level of the entire product.

Programmers refer to the Vendor, Tag, and Revision values as the *DMI triplet*.

When your code calls FFSTProbe, the DMI triplet values you specified must match the DMI values stored in the DMI database for your product. If the values do not match, FFSTProbe cannot find the VPD for your product in the DMI database.

If the DMI triplet for your product matches the DMI triplet of a different product, the results are unpredictable.

Deciding How and Where to Place Calls to FFSTProbe API

Here are two common approaches to instrumentation. One way is to place numerous calls to FFSTProbe throughout your product to get

broad coverage. This approach might contain only a minimum amount of error data since you know every error would be captured via a probe. The second approach is to use just a few strategically placed calls that capture greater amounts of error data to better isolate the the exact cause of the failure.

The advantage to the broad coverage approach is that errors are most likely to be identified because of the greater number of calls to FFSTProbe. The strategic approach usually involves instrumenting existing exception paths or thoroughly understanding the code to identify where to place the call to FFSTProbe.

You might consider combining both approaches in your code. The broad coverage aspect identifies exactly where the error occurred, and the strategic aspect identifies the cause.

You should use FFSTProbe only to detect problems that would require a program fix or a modification to user operation procedures.

Places to Instrument

The following list contains situations and places in your code you should consider for instrumentation:

- When your code generates an error return, create an error log entry for the error condition that caused the error.
- Some programmers use Print Debug and Print File for testing code. These instructions print certain variables and messages at various code failure points. Convert the Print Debug and Print File instructions to calls to the FFSTProbe API. The system disables the Print Debug and Print File functions after you install your code.
- When you expect return codes, create an error log entry when you receive unexpected return codes.
- In environment situations (circumstances that are not necessarily program errors but are worthy of creating an error log entry). For example, resource shortages, time-out conditions, system-hang conditions, or lost physical connections.
- In cleanup functions, your code may be tolerant of potential errors and may do some error recovery. The cleanup functions in your code are candidates for logging if the recovery signifies an important event.

Consider that the number of log entries and the size of entries you log could cause too much information to be logged. Creating excessive error log entries can cause the error log to wrap. This causes previously logged information to be overwritten. One of the most frequent questions that are asked about FFST is where and when to use it. When instrumenting your product, you should consider several places:

- Exception Paths

  Many programmers already take some actions in various exception conditions. These actions often include cleaning up execution environments, closing files, and ending the program. Your code should call the FFSTProbe API to create an error log entry that contains the following information:

  - the program or module that failed
  - why the failure occurred
  - what corrective actions to take.

- Incorrect Conditionals (for example, switch case)

  As developers write programs, they make assumptions of what can or cannot happen, and add various conditionals and execution blocks to programs.

  Conditionals that are not valid are ideal candidates for a call to the FFSTProbe API to log these failures. By calling the FFSTProbe function at these points, you can quickly and accurately pinpoint the failure and capture the associated data at the time of failure.

- External Calls

  OS/2 Warp Version 4 does not expect calls to external programs to fail. However, unexpected return codes, when not handled, can result in program failure. Your code should call FFSTProbe after each external call that results in an unexpected return code.

  Some development groups use someone other than the developer instrument all calls. Other groups spend more time anticipating the potential problem areas and placing probes only in those areas. Your code should do what is achievable for the current circumstances. You should then evaluate how well your calls to FFSTProbe work before you begin the next development cycle.

For FFSTProbe API calls to be useful in debugging a problem, the calls must specify:

- A unique probe ID

- Descriptive text explaining the problem
- Data that is relevant to the failure
- Instructions on how to resolve the problem if appropriate.

With well-instrumented code, several benefits of using the FFSTProbe function are evident. You can use FFST to capture error information. You can also identify areas in your code that **did not** cause the problem. If instrumented code made no calls to FFSTProbe, you can focus on code without calls to FFSTProbe.

Your code should not call the FFSTProbe function inside a loop. Call FFSTProbe only once per error situation. Repeated calls may cause system performance problems and cause wrapping of FFST data by storing unnecessary data.

Problem-prone components in products are good candidates for the FFSTProbe function.

Your decision about using instrumentation depends on the possible errors and the cost of solving an error.

Deciding What Data You Want to Collect

After you decide where to call the FFSTProbe function, you need to decide what data to capture. The question to ask is, "What data would I need to see to have a good chance of determining the source of the error?" Consider capturing data items that are global variables and control blocks.

Other things to consider are:

- How much data you need to determine the cause of the problem?
- Has this problem been encountered before?
- How complex is the code?
- Are other components or products being called?
- Does the error message information point to the problem?

Make every effort to collect enough data to solve the problem without requiring the user to re-create the problem.

The amount of data collected could also be affected by the amount of system storage that is available or allocated to store error data.

Error Types to Consider

When an error occurs, call the FFSTProbe function to log the error. The following examples describe several error types you should consider when you instrument your code and the types of data to capture for the error:

Error return

Determine the severity of an error so that you call the FFSTProbe function only when the error return indicates a serious problem. When a calling program has a significant failure that causes an error return, the program calls FFSTProbe. Be careful not to cause a "cascade" of calls to FFSTProbe as the system passes error returns back up through a set of higher level function calls.

Failure-related data may include:

- Return code
- Input parameters to the function
- Returned values from the function
- Any internal variables that determine or affect the erroneous results

Damaged data structures

Product data structures can become damaged with data that is not valid. To capture data for this type of error during normal processing, your code could have a method for periodically checking important internal data structures. Such logic is an important step toward improving the reliability and availability of the product.

Failure-related data may include:

- Data structures
- Historical information that indicates when your code found the product data structure to be correct
- General system data showing other programs in use by the system when the error occurred.

Time-outs and detected hangs

To detect time-outs and hangs, design your code to sense how long a given request should take.

Failure-related data may include:

- Current time-out values
- Any state information that describes what the timed-out function is currently doing
- States of resources that may relate to the time-out or hang
- Historical information that describes what the timed-out function had been doing before the error occurred.

Slow performance of a service

In order to detect a slow performance condition, design code to sense how long a given service should take before calling FFSTProbe.

Failure-related data may include:

- Internal resource states that may relate to the slow performance of the service
- Historical information that indicates who has been using that service and what requests the user made of the service.

Traps

It is difficult to detect a failure within a product and determine its cause after an exception management routine has received control. Well-designed and instrumented code can detect a failure before exception management routines get control.

Failure-related data may include:

- Exception blocks that contain the hardware state when the trap occurred
- Context data indicates what was being run when the trap occurred (for example, call stacks or internal state variables)

Ways to Collect Data

FFST takes care of storing the collected data in the error log and optional FFST dump. This information is available for viewing through use of the SYSLOG utility. For more information on error logs and the SYSLOG utility, see Viewing and Analyzing Error Log Entries.

User data could be data areas, control blocks, complete files, or any other form of data that could be used to determine the problem.

Two parameters on the FFST Probe function allow user-specified data to be collected:

- The *pDmpUsrData* parameter saves information in the FFST Dump. You can specify up to 30 items, each having a maximum size of 32 KB. For example, you use this parameter to save large control structures or buffers.
- The *LogUsrData* parameter save saves user data in the error log. The maximum amount of logged data is 2 KB. You need to determine what data you want as part of the 2 KB. For example, you use this parameter to save small items such as return codes, function names, or system names.

FFST Dump Data

The system creates FFST dumps when you use the *pDumpUserData* parameter with the FFSTProbe function. The *Enable FFST Dump* option on the FFST Probe Control Table Entry Summary window must be selected before the system will create a FFST dump (see Probe Control Table (PCT) Entry Add or Change Summary Window).

If you did not specify the parameter to collect the FFST dump in the original call to the FFSTProbe function, you can dynamically change the call. You use the Probe Control Table (PCT) to request the FFST dump the next time the specified call to FFSTProbe occurs. The system stores the FFST dump information in the file that is defined on the FFST Setup (FFSTCONF) window (see Using FFST Setup (FFSTCONF)). You can select the path but not the file name.

To delete FFST dumps, use the FFSTCONF command and select the *Actions* option. Then choose the *Dumps* option to display the FFST Dump File Summary window. Select the dump file to delete and click on the *File* menu bar option. Click on *Delete* to delete the dump file.

The FFST dump data can be of the various types. Some types may not be part of dump, depending on availability of data. The system displays the information when you format the dump. The various types are: process environment data, process status data, trace buffer data, user data, error log data, and process errors.

Process Environment Data

If you requested the process environment data, FFST collects and stores the data as part of the FFST dump. The system displays this information when you use the PM Dump Facility dump formatter.

You can specify to have the process environment data captured by selecting the *Capture Process Environment* checkbox on the FFST PCT Entry window (see Probe Control Table (PCT) Entry Add or Change Summary Window).

System Process Status Data

Process status data is a record of all processes and threads that are running on the system. This information is similar to information you get when you use the PSTAT command.

If you requested process status data, FFST collects and stores the data as part of the FFST dump. The system displays this information when you use the PM Dump Facility dump formatter.

You can specify to have the process status data captured by selecting the *Capture System Processes* checkbox on the FFST PCT Entry window (see Probe Control Table (PCT) Entry Add or Change Summary Window).

Trace Data

When you enable your code for trace, FFST collects and stores trace data. You can specify to have the trace snapshot captured by selecting the *Capture Trace Snapshot* checkbox on the FFST PCT Entry window (see Probe Control Table (PCT) Entry Add or Change Summary Window). The system stores trace information in a separate file.

You display trace information either by using the TRACEFMT command or by using the *Display Trace File* option in the SYSLOG Tools menu. For information about using the trace functions and formatter, see Analyzing Performance and Debugging Problems Using Trace.

User Storage Data

If you requested user storage data, FFST collects and stores the data as part of the dump. Using the function parameters, you can capture up to 30 data areas. The system displays this information when you use the PM Dump Facility dump formatter. For information about using the dump functions and formatter, see Capturing and Saving Failure-Related Information through Dumps.

Additional Error Log User Data

If you requested additional error log user data in the FFSTProbe call, FFST generates a FFST dump and stores the data as part of the dump. This information is identical to the error log entry that is stored in the error log. The system displays this information when you use the PM Dump Facility dump formatter. For information about using the error log functions and formatter, see Viewing and Analyzing Error Log Entries.

FFST Dump Process Errors

FFST collects and stores FFST dump-processing errors that occurred when the system creates the dump. The system includes error message identifiers in this information. The system displays this information when you use the PM Dump Facility dump formatter.

Process Dumps

FFST collects and stores a process dump only when the *Capture process dump* option is selected on the Probe Control Table (PCT) Entry Summary window. Refer to FFST Probe Control Table Entry Summary Window.

System Dumps

FFST collects and stores a system dump only when the *Capture system dump* is selected on the Probe Control Table (PCT) Entry Summary window. Refer to FFST Probe Control Table Entry Summary Window.

Coding the FFSTProbe Functions

You can code the FFSTProbe function to capture data in several ways:

- By direct call to the FFSTProbe function
- By coding a macro that uses FFSTProbe
- By creating a subroutine to call the FFSTProbe function if you specify certain parameter values.

The best way to call the FFSTProbe function is through the subroutine method.

Direct Calls

Your code calls the FFSTProbe function at each specific instrumentation point. All pertinent parameters need to be specified on each call to capture the required data.

As you can see, when you use several calls to the FFSTProbe function, you greatly increase your coding efforts. You need to specify every parameter for each call in order to capture the required error data for problem analysis. This method requires you to change each call to FFSTProbe in your code if you decide to change parameters and is difficult to maintain.

Using Macros to Call the FFSTProbe Function

You can also use a macro to call the FFSTProbe function; however, this is not recommended. When you use a macro, the macro gets expanded into working code when you compile your code. Using a macro causes your components to increase in size. Macros could have a dramatic effect on the size your product if your code has numerous calls to FFSTProbe.

Using Subroutines to Call the FFSTProbe Function

An effective way to instrument a call to FFSTProbe is to create a subroutine in your code components. Many FFSTProbe parameters for your code contain the same values for each call. Using the subroutine allows you to code these parameters once rather than requiring all parameter information for each individual call as in the direct call approach.

Using the subroutine provides common parameter information for all calls to FFSTProbe in your code. When you need to change parameter information, you only change the parameters in the FFST subroutine rather than the parameters on every call to FFSTProbe.

See Example of an Application Program Using a Subroutine for an example of how you use the subroutine method to call the FFSTProbe function.

Creating Template Files

Use the MKTMPF (Make Template File) command to create template files with message IDs that refer to text information about the error. Use a text editor to create an input file for your template. Input files may have any file name that is valid for your file system.

SYSLOG uses template files to format and display error data and user data in the error log entry. There should be a unique template file entry for each template ID used by your code.

For information regarding the template files and MKTMPF command, see the documentation in the *OS/2 Warp Version 4 Tools Reference* located in the Toolkit.

Why Template Files Are Important

In the template file are the identification numbers of the error messages and possible recovery actions for the error messages. The template file also contains the formatting instructions for LogUsrData entries in the system error log. Each template in the file has a unique identifier that is specified in the parameters of the call to the FFSTProbe function.

The system creates an error log entry that contains the information that is specified on the call to FFSTProbe. The system also stores the VPD information that is related to the calling product in the error log entry.

When you instrument your code, ensure that an appropriate error record template entry exists in DMI for your product. The FFSTProbe function needs to point to a specific error record template entry within a the template file. Each product should maintain a separate error record template file.

See Creating an Error Record Template Input File for an example of an error record template input file.

Creating Message Files

Message files contain the text of the error messages that error log entries use. The process of producing message files begins with creating an input file. The contents of the input file define the product the messages belong to, the message numbers, and text of the messages. The message text can include the substitution variables provided by FFSTProbe's MSGINSDATA parameter. After you complete the input file, use MKMSGF to compile the file. The output of the compiler is the message file.

A description of how you create an input file and use MKMSGF to create the message text file is in the Toolkit *OS/2 Warp Version 4 Tools Reference* .

See Message Input File Example for an example of a message input file.

Setting Up (Instrumenting) for Trace

OS/2 Warp Version 4 applications can be instrumented for trace by adding calls to the TraceCreateEntry function to collect user-specified data. When running the program, the system collects specified data and stores the data in a trace file. Use the trace format (TRACEFMT) command discussed Analyzing Performance and Debugging Problems Using Trace to view and analyze the data stored in the trace file.

Using trace points is one of the best ways to collect serviceability information on a customer system. It allows service personnel and developers to follow the course of events that lead to the failure.

What Is Trace?

Trace is a tool that you use for performance analysis and service level debugging. Use the trace tool after you perform development and debugging at the individual module level or object level. Microkernel, operating system, and application programmers as well as service personnel can use the trace facilities to assist in tracking code problems. For more information about using trace, refer to Analyzing Performance and Debugging Problems Using Trace.

Creating a Trace File Entry Using the TraceCreateEntry Function

Use the TraceCreateEntry function to create an entry in a trace file when trace is being performed.

The TraceCreateEntry function uses a single parameter packet structure, TCEREQUEST, that is defined in the *trace.h* file that is part of the Toolkit. Use the packet to specify the following information:

- Major and minor codes
- An optional set of state-related data that is to be logged within the trace entry that the trace point creates

A parameter packet is a structure that contains values or parameters.

The TraceCreateEntry caller does not have to provide any logic to check whether you turned on the trace point. The TraceCreateEntry function performs all necessary checking.

Note: Some additional overhead exists when you use the TracecCreateEntry function without checking the state of the trace first. The trace service automatically time-stamps all trace entries that TraceCreateEntry creates.

For more information about these and other trace related functions, refer to Problem Determination APIs.

In the example that is described in Examples of Code when Instrumenting for FFST and Trace, the code calls the TraceCreateEntry function. This example logs two data variables for the trace point. The first variable is a 32-bit status code. The second variable is a 16-bit flag word:

An application program can add trace points to a program. The program adds trace points by using the TraceCreateEntry function. Refer to Problem Determination APIs for information about the TraceCreateEntry function.

Defining Trace Information Format

You create the defined format for a major code by using TRCUST. The system formats trace entries to represent the actual major code by descriptive text instead of just numbers. In addition, the system can format data saved by the trace entries instead of being displaying the data as hexadecimal numbers. The use of TRCUST is completely optional.

The trace source file (.TSF) contains lines that describe the format of each major and minor code. TRCUST uses the .TSF file to generate .TFF files (trace format files). TRCUST generates one .TFF file for each major code. The trace formatter program uses the .TFF files to determine how to format the trace point entries being displayed.

Creating Trace Entry Formatting Directives

To define trace entry formatting directives, you should place entries in a Trace Source File (.TSF) file. The .TSF file is an ASCII text file that defines the trace points being traced in the program. The TRCUST Trace Point Definition tool uses the .TSF file. TRCUST converts the information in the .TSF file into Trace Format Files (.TFF) having the name of TRC00xxx.TFF, where *xxx* is the major code defined in the .TSF file. The trace formatter uses the TFF file to format and displays the trace information.

The following file **TEST.TSF** is an example of a simple .TSF file. You use the .TSF file to specify the formatting directives for a trace entry that TraceCreateEntry creates:

TEST.TSF Trace Source File Example

```
MODNAME = probe.exe
MAJOR = 220

TRACE MINOR=1,
  TP=@STATIC,
  DESC="Tracepoint example start",
  FMT ="Data %D String %S"

TRACE MINOR=0x8001,
  TP=@STATIC,
  DESC="Tracepoint example end",
  FMT ="Data %D String %S"
```

For more information about the TRCUST command, the .TSF file and syntax used in the file, refer to the *Trace Customizer User's Guide* in the Toolkit.

Creating DMI MIF Files

The Desktop Management Interface (DMI) Management Interface Format (MIF) file is an ASCII text file that contains the attributes that describe manageable products. The MIF defines a standard way of providing the product information to be retrieved by product management applications.

A manageable product provides information about itself in the form of attributes in a MIF file that is stored in the MIF database. There is a MIF file for each manageable product on the system.

You use the attributes in the MIF file to describe components and characteristics of your product. Attributes can define the component version and revision date, the date you installed the product, or the hardware requirements that are needed to use your product. Group the specific attributes together. In this example, the component version and the revision date would be one group in the MIF file, and hardware requirements would be in a different group.

Management applications retrieve the information in your MIF file. You use the management applications to list information about your specific manageable product or all manageable products that are installed on a system.

The management applications, such as SystemView Agent or DMI Browser, contain functions that can change the DMI information for a specific manageable product. You can change the DMI information of a manageable product without reinstalling the product. The product monitors its own DMI information for any changes. When changes occur, the product updates the functions that it performs based on the new DMI information.

You can find the MIF file requirements in the *SystemView Agent Programmer's Guide* in the Toolkit in the Problem Determination Tools folder.

There is an example MIF file in the Toolkit that you can copy as a base for the MIF file for your product. The file name is **PROBEMXP.MIF**.

For an example of a MIF file, see MIF File Example.

How to Install Your Software Product

You can use the software installer to install your product and the associated MIF file. The *Software Install Reference* documents the steps required to install your product and MIF file. This document is in the Software Install Package in the Software Developer Kit and is also available on The Developers Connection for OS/2 compact disk.

The *Getting Started with Software Install* in the Software Developer Kit explains how to install and use the Software Developer Kit tools.

For more information about DMI and MIF, refer to the The Desktop Management Interface (DMI).

FFST-Related Functions

You can change the configuration information with a program by using the FFSTQueryConfiguration and FFSTSetConfiguration functions in your code.

When you call FFSTQueryConfiguration, the system returns the configuration information into a structure that you specify. Your code then compares the configuration information in the structure to the configuration values your code is expecting. If any of the values are different, your code can change them by calling the FFSTSetConfiguration function. You then specify the configuration information for the system to use.

Refer to Problem Determination APIs for more information about these functions.

Examples of Code when Instrumenting for FFST and Trace

This section contains the following examples:

- how you instrument a product by using subroutines to call FFSTProbe and TraceCreateEntry
- a template input file
- a message input file
- a MIF file

Example of an Application Program Using a Subroutine

This example shows the user application PROBE.C. The application uses the subroutine approach to call the FFSTProbe function.

User Application using FFST and Trace Subroutines

```
/*********************************************************************/
/* probe.c: FFSTProbe sample                                         */
/*                                                                   */
/* This test program gives an example of using the FFSTProbe API and the  */
/* TraceCreateEntry API by using 'wrapper' functions.  The dummy API      */
/* My_Dummy_Api returns a return code which is then used as the basis of  */
/* firing a FFSTProbe via the wrapper function, callFFST.  callFFST can   */
/* be modified to include more or less data as needed.               */
/*                                                                   */
/*********************************************************************/

#define INCL_DOS
#define INCL_DOSMEMMGR
#define INCL_DOSPROCESS
#define INCL_FFST
#define NO_ERROR 0

#include <os2.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <FFST.h>
#include <trace.h>

/*****************************************************************************/
/* Define probe ID for FFSTProbe called when dummy API fails.  Probe ID      */
/* is the unique identifier you use later to find the source of the          */
/* failure.  It should be unique within a DMI triplet (explained later)      */
/* or within your product                                                    */
/*****************************************************************************/
#define DUMMY_API_PROBE 22222

void callFFST ( ULONG input_version,           /* FFST 'Wrapper' Function */
            /* input version lets you change the wrapper     */
            /* without changing each call, just make sure    */
            /* the wrapper still treats the 'old' version    */
            /* the same and that any new code is             */
            /* conditioned on a new input_version #          */

                ULONG input_probe_flags,         /* FFSTProbe probe flags */
                ULONG input_severity,               /* FFSTProbe severity */
                ULONG input_probe_id,                     /* FFSTProbe ID */
                CHAR* input_module_name,   /* module name passed to probe */
                ULONG input_log_data_length, /* log data length for the
                                                system error log          */
                PVOID input_pError_log_data,  /* pointer to the data for
                                                 system error log         */
                int   argc);

/*****************************************************/
/* This is for a common trace entry routine          */
/*****************************************************/
#define HKWD_TEST            220        /* major code */
#define hkwd_test_entry      0x0001     /* minor code for entry */
#define hkwd_test_exit       0x8001     /* minor code for exit */

struct
{
  int count;
  char textì12 ;
} trace_capture_start, trace_capture_end;

APIRET trace_out(ULONG major, ULONG minor, void *trace_data,
                 ULONG data_len);      /* trace wrapper function */

/*************************************/
/* End of trace declarations for Main */
/*************************************/

ULONG My_Dummy_API(ULONG Mydata);


/*****************************************************************************/
/*                                                                           */
/*  Main Application (this uses the callFFST wrapper function).              */
/*                                                                           */
/*****************************************************************************/

int main ( int argc, char * argvì , char * envp  )
{
   ULONG  rc          = 0;
   ULONG  Mydata      = 2;
   ULONG  userDataLen = 0;
   PVOID  pUserData   = NULL;

   printf ( "Starting FFSTProbe Sample\n" );

/*************************************/
/* Do the trace entry point           */
/*************************************/

   trace_capture_start.count = 3; /* just a number */
   strncpy(trace_capture_start.text, "Start main", 12);

/******** CALL TraceCreateEntry function ****/
   trace_out(HKWD_TEST,
      hkwd_test_entry,
      &trace_capture_start,
```

```
      sizeof(trace_capture_start));

   /*********************************************************************/
   /* call the 'dummy'  API so it returns a non-zero rc                 */
   /*********************************************************************/
   rc = My_Dummy_API ( Mydata );
   if ( rc != NO_ERROR )
   {
      /*******************************************************************/
      /* The API has failed. Setup the userData to contain the failing rc */
      /*******************************************************************/
      pUserData = calloc ( 2, sizeof ( ULONG ) );
      memcpy ( pUserData, &rc, sizeof ( ULONG ) );
      memcpy ( ( PBYTE ) pUserData + sizeof ( ULONG )
             , &Mydata, sizeof ( ULONG ) );
/*************************************************************************/
/* Call the FFSTProbe wrapper function with a version of 1,              */
/* Have FFST post the process status and environment variables in the    */
/* syslog, a severity of 4, a probe id of DUMMY_API_PROBE which was      */
/* previously defined as 22222, a module name of 'my_module_1', the length */
/* of logusrdta, the logUserData (equal to the failing rc (1) as         */
/* setup above) and Argc is passed in to determine whether or not data   */
/* should be retrieved from DMI.                                         */
/*************************************************************************/


         callFFST ( 1
               , PSTAT_FLAG   PROC_ENV_FLAG
               , SEVERITY4
               , DUMMY_API_PROBE
               , "my_module_1"
               , 2 * sizeof ( ULONG )
               , pUserData
               , argc );
   }

   if (pUserData != NULL)
   {
     free(pUserData);
     pUserData = NULL;
   }

   if (argc > 1)
   {
      printf("\nFFSTProbe sample ended not using DMI component:\n\n\n");
   }
   else
   {
      printf("\nFFSTProbe sample ended using DMI component:\n\n\n");
   }

/*************************************/
/* Do the trace end point            */
/*************************************/

   trace_capture_end.count = 99;
   strncpy(trace_capture_end.text, "End main", 12);

/******** CALL TraceCreateEntry function ****/
   trace_out(HKWD_TEST,
      hkwd_test_entry,
      &trace_capture_end,
      sizeof(trace_capture_end));

   return 0;
}



/*************************************************************************/
/* callFFST is the FFSTProbe wrapper function. It allows you to code the  */
/* FFSTProbe API once with data that is static as far as your usage is    */
/* concerned and allows you to pass in dynamic data.   It also helps      */
/* insulate your code if you decide to change your 'static' options       */
/*************************************************************************/

void callFFST ( ULONG input_version,           /* FFST 'Wrapper' Function */
                ULONG input_probe_flags,         /* FFSTProbe probe flags */
                ULONG input_severity,               /* FFSTProbe severity */
```

```
                ULONG input_probe_id,                    /* FFSTProbe ID */
                CHAR* input_module_name,   /* module name passed to probe */
                ULONG input_log_data_length, /* log data length for the
                                               system error log         */
                PVOID input_pError_log_data,   /* pointer to the data for
                                                  system error log       */
                int   argc)
{
  APIRET  rc = 0;
  PVOID   pvar_n0;
  ULONG   pvar_n1;

  /*********************************************************************/
  /* FFSTProbe API structures.    Described in the API Guide           */
  /*********************************************************************/
  FFSTPARMS     FFSTParms;
  PRODUCTINFO   productInfo;
  PRODUCTDATA   productData;
  DMIDATA       DMIData;
  DUMPUSERDATA  dumpUserData;
  MSGINSDATA    msgInsData;

  /*********************************************************************/
  /* The PRODUCTDATA structure defines the DMI triplet which allows    */
  /* additional product information, including template repository     */
  /* filename, to be retrieved from DMI.   DMI is a industry standard  */
  /* for desktop mgt                                                   */
  /*********************************************************************/
  productData.packet_size            = sizeof ( productData );
  productData.packet_revision_number = PRODUCTDATA_ASCII;
                                       /* data can be ASCII or UNI */
  productData.DMI_tag                = "FFSTProbe Sample";
                                      /* Customize for your program */
  productData.DMI_vendor_tag         = "IBM";
                                       /*Customize for your company */
  productData.DMI_revision           = "1.00";            /* Customize  */

  /*********************************************************************/
  /* The DMIDATA structure below is the information which can either be  */
  /* retrieved by DMI or passed in by the FFSTProbe function. The      */
  /* preferred method is to use DMI. In the example below, you can see   */
  /* the use of either depending on whether or not a parm was passed on  */
  /* call to this program                                              */
  /*********************************************************************/

  if ( !(argc > 1) )
  {
     /*******************************************************************/
     /* Setting this structure to NULL indicates that the information is */
     /* to be retrieved from DMI using the DMI triplet as defined in the */
     /* productData structure.  This is the preferred method.           */
     /* Other files in this example show how to build your own DMI      */
     /*******************************************************************/
     productInfo.pDMIData = NULL;

     /*******************************************************************/
     /* Note: This shows the usage of message insert text and is NOT part*/
     /* of the information that could or could not be retrieved from DMI */
     /* This is included as an example of MsgInsTxt and how it can be    */
     /* used to send probe specific data to the SYSLOG (System Error Log)*/
     /*******************************************************************/
     msgInsData.MsgInsTxti0 .insert_number = 1;
     msgInsData.MsgInsTxti0 .insert_text   = "We did use a DMI component";
  }
  else
   {
     /*******************************************************************/
     /* fill the DMI data structure - useful only in test environments   */
     /*******************************************************************/
     DMIData.packet_size              = sizeof ( DMIData );
     DMIData.packet_revision_number   = DMIDATA_ASCII;
                                        /* could be unicode instead */
     DMIData.DMI_product_ID           = "FFST_toolkt_sample";
                                      /* note this is different than tag */
     DMIData.DMI_modification_level   = "000000";
     DMIData.DMI_fix_level            = "010101";
     DMIData.template_filename        = "PROBE.REP";
                                       /* this file must be on the DPATH */
     DMIData.template_filename_length = strlen (DMIData.template_filename)
```

```
                                          * sizeof ( char );
                                       /* since ascii is being used */
     productInfo.pDMIData            = &DMIData;

     /*********************************************************************/
     /* Note: This shows the usage of message insert text and is NOT a    */
     /* of the information that could or could not be retrieved from DMI */
     /*********************************************************************/
     msgInsData.MsgInsTxti0 .insert_number = 1;
     msgInsData.MsgInsTxti0 .insert_text   = "We did not use a DMI component";
    }

   /*********************************************************************/
   /* set the pointers up for PRODUCTINFO                               */
   /*********************************************************************/

   productInfo.pProductData = &productData;   /* This points to the DMI
                                                 related data          */

   /*********************************************************************/
   /* set up some DUMPUSERDATA items                                    */
   /*********************************************************************/
   pvar_n0 = "Dump user data";             /* Anything can be dumped
                                              here up to 32 Kbytes     */
   pvar_n1 = 2;
   dumpUserData.no_of_variables = 2;
   dumpUserData.DumpDataVari0 .var_n_length = strlen(pvar_n0) + 1;
   dumpUserData.DumpDataVari0 .var_n        = pvar_n0;
   dumpUserData.DumpDataVari1 .var_n_length = sizeof(ULONG);
   dumpUserData.DumpDataVari1 .var_n        = (PVOID)(&pvar_n1);

   /*********************************************************************/
   /* set up a couple of MSGINSDATA messages- just to show it can be done */
   /*********************************************************************/
   msgInsData.no_inserts   = 2;
   msgInsData.MsgInsTxti1 .insert_number = 2;
   msgInsData.MsgInsTxti1 .insert_text   = "Message insert variable 2";

   /*********************************************************************/
   /* set the FFSTPARMS structure, most values from DEFINEs above.      */
   /* See API GUIDE for details on each field and their possible values */
   /*********************************************************************/
   FFSTParms.packet_size            = sizeof ( FFSTParms );
   FFSTParms.packet_revision_number = FFSTPARMS_OS2_ASCII;
                                           /* ASCII vs UNICODE data */
   FFSTParms.module_name            = input_module_name;
   FFSTParms.probe_ID               = input_probe_id;
   FFSTParms.severity               = input_severity;
   FFSTParms.template_record_ID     = input_probe_id;
   FFSTParms.pMsgInsData            = &msgInsData;
   FFSTParms.probe_flags            = input_probe_flags;
   FFSTParms.pDumpUserData          = &dumpUserData;
                                    /* dump data is stored in .DMP files */
   FFSTParms.log_user_data_length   = input_log_data_length;
   FFSTParms.log_user_data          = input_pError_log_data;
                       /* log data is stored as part of the SYSLOG entry */

   /*********************************************************************/
   /* Call the FFSProbe API                                             */
   /*********************************************************************/
   if ( input_version == 1)
   {
      rc = FFSTProbe ( &productInfo, &FFSTParms);
   }

   printf("\n----- Fired the FFSTProbe, rc=%d\n",rc);
               /* for example only, do not do this in customer level code */

}

/*********************************************************************/
/*  This is the dummy  API for use in the example.  It can easily set     */
/*  non-zero rc's                                                    */
/*********************************************************************/

ULONG My_Dummy_API ( ULONG Mydata )
{
   if ( Mydata != 123456 )
   {
```

```
      return 1;
   }
   else
   {
      return 0;
   }
}

/*********************************************************************/
/*  Trace events function                                            */
/*********************************************************************/
APIRET trace_out(ULONG major, ULONG minor, void *trace_data, ULONG data_len)
{
  TCEREQUEST packet;
  APIRET      rc;

  packet.packet_size = sizeof packet;      /* Size of packet in bytes       */
  packet.packet_revision_number = TRACE_RELEASE; /* Revision level of trace */
  packet.major_event_code = major;         /* Major code event to be logged */
  packet.minor_event_code = minor;         /* Minor code event to be logged */
  packet.event_data_length = data_len;     /* Length of callers event buffer*/
  packet.event_data = trace_data;          /* Pointer to callers buffer     */

  /* call the TraceCreateEntry function */
  rc = TraceCreateEntry(&packet);
  return rc;
}
```

Creating an Error Record Template Input File

You use a text editor to create an input file. MKTMPF uses this file to create error record template files. There should be one set of default path names and one or more sets of template entries for each use of MKTMPF.

The following example shows a template input text file to use with the MKTMPF command for the procedure in the PROBE.C example.

Error Record Template Input File

```
Descriptive_Name         = 'System Management Example Template file'
*
* 'File name only' can be specified as long as the message files are on the
* system's DPATH otherwise full path must be specified.
* NOTE:  MESSAGE FILES WILL BE 6 CHAR file name with a .MSG Extension
* Created by MKMSGF utility
*
* A single template file can contain entries for an entire product or a
* single program *depending solely on the development teams preferences.
* The only requirement is that all users of the template file use the
*same DMI triplet or hardcode to the same path
*
Default_message_pathname = probe.msg
Default_causes_pathname  = probe.msg
Default_actions_pathname = probe.msg
Default_details_pathname = probe.msg
*
* Template number is associated with the FFSTProbe call
*
Template_number          = 11111
*
* Message_number is the message number you want displayed on the SYSLOG
* summary screen
*
Message_number           = 1
*
* Log_class 1 is hardware, 2 is software
*
Log_class                = 2
*
```

```
* Default_xxxxx_pathname can be overridden on a template entry basis
* like this:
* Causes_pathname          = c:\os2\system\othrCmsg.msg
* Actions_pathname         = c:\os2\system\othrAmsg.msg
* Details_pathname         = c:\os2\system\othrDmsg.msg
*
* On the right of the = sign you specify which message number you want
* displayed in the relevant _causes _actions section of the SYSLOG
* display. Here message 4 is displayed from PROBE.MSG in the
* Fail_causes section of SYSLOG.
*
Fail_causes              = 4,0,0,0
Fail_actions             = 0,0,0,0
Install_causes           = 0,0,0,0
Install_actions          = 0,0,0,0
User_causes              = 0,0,0,0
User_actions             = 0,0,0,0
*
Template_number          = 22222
Message_number           = 2
Log_class                = 2
Fail_causes              = 4,0,0,0
Fail_actions             = 0,0,0,0
Install_causes           = 0,0,0,0
Install_actions          = 0,0,0,0
User_causes              = 5,0,0,0
User_actions             = 6,0,0,0
*
* Detail data allows you to format the FFSTProbes log_user_data (if
* used). You can use as many Detail data phrases as needed.  The
* format is:  Detail_data = L,O,H,T where L is Length of data, O is the
* Offset it starts at, T is the Type of data as it was sent and how you
* want it formatted (e.g. binary show as hex) and H is the heading
* message number you want to display with it.  When your run the example
* you will notice the difference in formatting.
*
Detail_data              = 4,0,7,3
Detail_data              = 4,4,9,2
*
```

Template File Tips

The following list contains other helpful tips about the error record templates:

- Template files use message-file path names without drive letters. You must ensure the following:
    - you place the message files in the proper directory
    - you place the directory in the DPATH statement of CONFIG.SYS
    - the template entries in the file point to the proper message file.
- The message resource files (xxx.mes) that are specified in the templates are created by using MKMSGF. You install these files on the system during product installation.
- The error logging facility directly accesses the error record template files to find error messages, causes of the error, and corresponding recovery actions in various message files. The system stores this information in a file in the form of message identifier numbers and message file path names.

Message Input File Example

You use a text editor to create an input file. MRES utility uses this file to create message files. The following example shows what you can entered in a message text file. You use the input file with MKMSGF to create messages.

Message Input File

```
;Input file for MKMSGF the resulting .MSG file must be placed in the DPATH or
;the template file specification must specify the path on the boot drive.
;
;The 'number' component of the msg id must be 4 numeric chars.
;This 'number' is what is used in the template file _causes _actions phrases
;between the comma's. e.g. Fail Causes = 4,5,0,0 will cause SYSLOG to display
;message BNB0004I followed by BNB0005I in its fail causes section.
;Message text can continue across multiple lines if so desired.
BNB
BNB0001I: Message for probe 11111, with 2 substitution vars: %1 and %2
BNB0002I: Message for probe 22222, with 2 substitution vars: %1 and %2
BNB0003I: Hello World!
BNB0004I: Hello World! My_Dummy_API returned a non-zero rc
BNB0005I: A non-zero rc is the result of 123456 not being sent to My_Dummy_API
BNB0006I: Call My_Dummy_API with 123456 passed as Mydata.
BNB0007I: Is the Log User Data sent with this probe, formatted to hexidecimal
BNB0008I: Is the Log User Data sent with this probe, formatted to ASCII
BNB0009I: Is the Log User Data sent with this probe, formatted as decimal
```

MIF File Example

This is an example of the MIF file for the sample program. You can find this file in the Toolkit. The file name is PROBEXMP.MIF.

MIF File Example

```
  start component
      name        = "FFST Example code"
      description = "This is the FFST Example code Component"
  //                                                                  //
  // This is a particularly complicated file.  Fortunately you only need  //
  // to deal with the section at the very bottom, where you put in the    //
  // information related to your DMI triplet (Vendor Tag, Tag, and        //
  // Revision. The rest of this file defines itself to DMI and layouts    //
  // the fields of information that DMI will manage                       //
  //                                                                      //
  //////////////////////////////////////////////////////////////////////////
  //                                                                  //
  //                      component id group                          //
  //                                                                  //
  //////////////////////////////////////////////////////////////////////////
  // Component ID Group

  //
  //   Enumerations for this group
  //
             Start Enum
               Name = "Verify_Type"
               Type = Integer
               0x00 = "An error occurred; check status code"
               0x01 = "This component does not exist"
               0x02 = "The verify is not supported"
               0x03 = "Reserved"
               0x04 = "This component exists, but the functionality is untested"
               0x05 = "This component exists, but the functionality is unknown"
               0x06 = "This component exists, and is not functioning correctly"
               0x07 = "This component exists, and is functioning correctly"
             End Enum


      start group
          name    = "ComponentID"
          id      = 1
          class   = "DMTF ComponentID 1.0"
          description = "This group defines attributes common to all
                         components.  This group is required."
```

```
        start attribute
            name    = "Manufacturer"
            id      = 1
            description = "The name of the manufacturer that
                           produces this component."
            access  = READ-ONLY
            storage = COMMON
            type    = STRING(64)
            value   = "IBM Corp."
        end attribute

        start attribute
            name    = "Product"
            id      = 2
            description = "The name of the component."
            access  = READ-ONLY
            storage = COMMON
            type    = STRING(64)
            value   = "FFST Example code"
        end attribute

        start attribute
            name    = "Version"
            id      = 3
            description = "The version for the component."
            access  = READ-ONLY
            storage = COMMON
            type    = STRING(64)
            value   = ""
        end attribute

        start attribute
            name    = "Serial Number"
            id      = 4
            description = "The serial number for this instance of
                           this component."
            access  = READ-ONLY
            storage = SPECIFIC
            type    = STRING(64)
            value   = ""
        end attribute

        start attribute
            name    = "Installation"
            id      = 5
            description = "The time and date of the last install of
                           this component."
            access  = READ-ONLY
            storage = SPECIFIC
            type    = DATE
            value   = ""
        end attribute

        Start Attribute
            Name = "Verify"
            Id = 6
            Access = Read-Only
            Storage = Specific
            Type = "Verify_Type"
            Description = "A code that provides a level of verification "
                          "that the component is still installed and working."
            Value = 0x07
        End Attribute

    end group

//
//  Software Product Group
//
//      This is the group that contains the software Vital Product
//      Data attributes that identify the software product that is
//      represented by this DMI component
//
//      This group is a template that defines the software product group
//
//      This definition should not be modified the actual values are found in
//       the table which is after this template definition
//
//      There are two sample rows defined for the table, however the table can
```

```
//       have one or more rows, what ever is correct for your component
//

    start group
        name    = "OS/2 Warp Software Product"
        class   = "IBM_OS/2 Warp SoftwareProduct 1.0"
        description = "OS/2 Warp standard Software Product attributes"
        key=2,4,5              // The (Tag, VendorTag, Revision) triplet acts
                                  //   as the key to this group

        start attribute
            name    = "Title"
            id      = 1
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Long name for software product package unit"
        end attribute

        start attribute
            name    = "Tag"
            id      = 2
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Short name for software product package unit"
        end attribute

        start attribute
            name    = "VendorTitle"
            id      = 3
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Long name for manufacturer of software
                           product package unit"
        end attribute

        start attribute
            name    = "VendorTag"
            id      = 4
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Short name for manufacturer of software
                           product package unit"
        end attribute

        start attribute
            name    = "Revision"
            id      = 5
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Major, Minor and Modification levels for software
                           product package unit"
        end attribute

        start attribute
            name    = "ModificationLevel"
            id      = 6
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Current CSD level for software product
                           package unit"
        end attribute

        start attribute
            name    = "SelectiveFixLevel"
            id      = 7
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Selective Fix level for software product
                           package unit"
        end attribute

        start attribute
```

```
            name    = "Description"
            id      = 8
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "String that describes the software product
                          package unit"
        end attribute

        start attribute
            name    = "ParentTag"
            id      = 9
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Short name for the parent package unit of this
                          software product package unit"
        end attribute

        start attribute
            name    = "ParentVendorTag"
            id      = 10
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Short manufacturer name for the parent package unit
                          of this software product package unit"
        end attribute

        start attribute
            name    = "ParentRevision"
            id      = 11
            access  = READ-ONLY
            type    = STRING(256)
            value   = "     "
            description = "Short revision string for the parent package unit
                          of this software product package unit"
        end attribute

        start attribute
            name    = "Error Record Template File"
            id      = 12
            access  = READ-ONLY
            type    = STRING(508)
            value   = "     "
            description = "Name of the Error Record Template file for this
                          software product package unit"
        end attribute

    end group

//
//  Software product group table with actual values
//
//      This is the group that contains the software Vital Product
//      Data attributes that identify the software product that is
//      represented by this DMI component
//
//      There are two sample rows defined for the table, however the table can
//       have one or more rows, what ever is correct for your component
//
//       Note each of the fields assigned a position value below
//       are defined in the sections above.
//
//       THIS IS THE ONLY SECTION YOU NEED TO DEAL WITH FOR THE PURPOSE OF
//       BUILDING YOUR OWN EXAMPLE
//
    start table
        name = "Software components and their VPD"
        id = 2
        class   = "IBM_OS/2 Warp SoftwareProduct 1.0"

        {"",
         "FFSTProbe Sample",       //Tag - Part of the triplet,
                                   //    identifies your product
         "",
         "IBM",                    // Vendor Tag - Part of the triplet,
                                   //    identifies your company
         "1.00",                   //  Revision - Part of Triplet
```

```
          "000000",                // Modification Level - Specified in your
                                   //    code if not looked up
          "010101",                // Selective Fix Level - Specified in your
                                   //    code if not looked up
          "FFSTProbe Example Program", // Description
          "",
          "",
          "",
          "PROBE.REP"}             // Repository (Template File) Name.
                                   // Should be on your DPATH
                                   // Otherwise full path would be specified.
                                   // This field is the prime
                                   // reason FFSTProbe uses DMI. The triplet
                                   // above forms the key
                                   // to retrieve the repository file
                                   // name so that the probe can be
                                   // formatted by SYSLOG

     end table

  end component
```

Controlling FFSTProbe Calls

After you install your code on the system, you can choose to capture data other than that specified on the call to the FFSTProbe function. This section describes how to temporarily change the values that FFSTProbe uses while your code is running. This eliminates the need to change the parameter values and recompile your code.

Controlling FFSTProbe

First Failure Support Technology (FFST) provides graphical user interfaces (GUIs) that are used to control how the FFSTProbe function captures information.

You use the FFST commands or the user interface icons in the Problem Determination Tools folder.

FFST provides the following user interfaces:

- FFST Setup icon
- FFSTCONF command

You can view and change the FFST setup information by using the FFSTCONF command. You can also update and view probe control table (PCT) information through the setup GUI. You can control probes from the SYSLOG utility by using the **Tools** menu option from any details window and selecting the **Modify Entry Collection** option.

Using FFST Setup (FFSTCONF)

You can use the FFST setup utility to define the following configuration information:

- where to store dump files
- total maximum size for all FFST files in kilobytes (KB)
- whether you allow files to be wrapped
- whether you allow files to be appended.

The FFST setup contains the following configuration information:

- Information that is required for FFST data collection

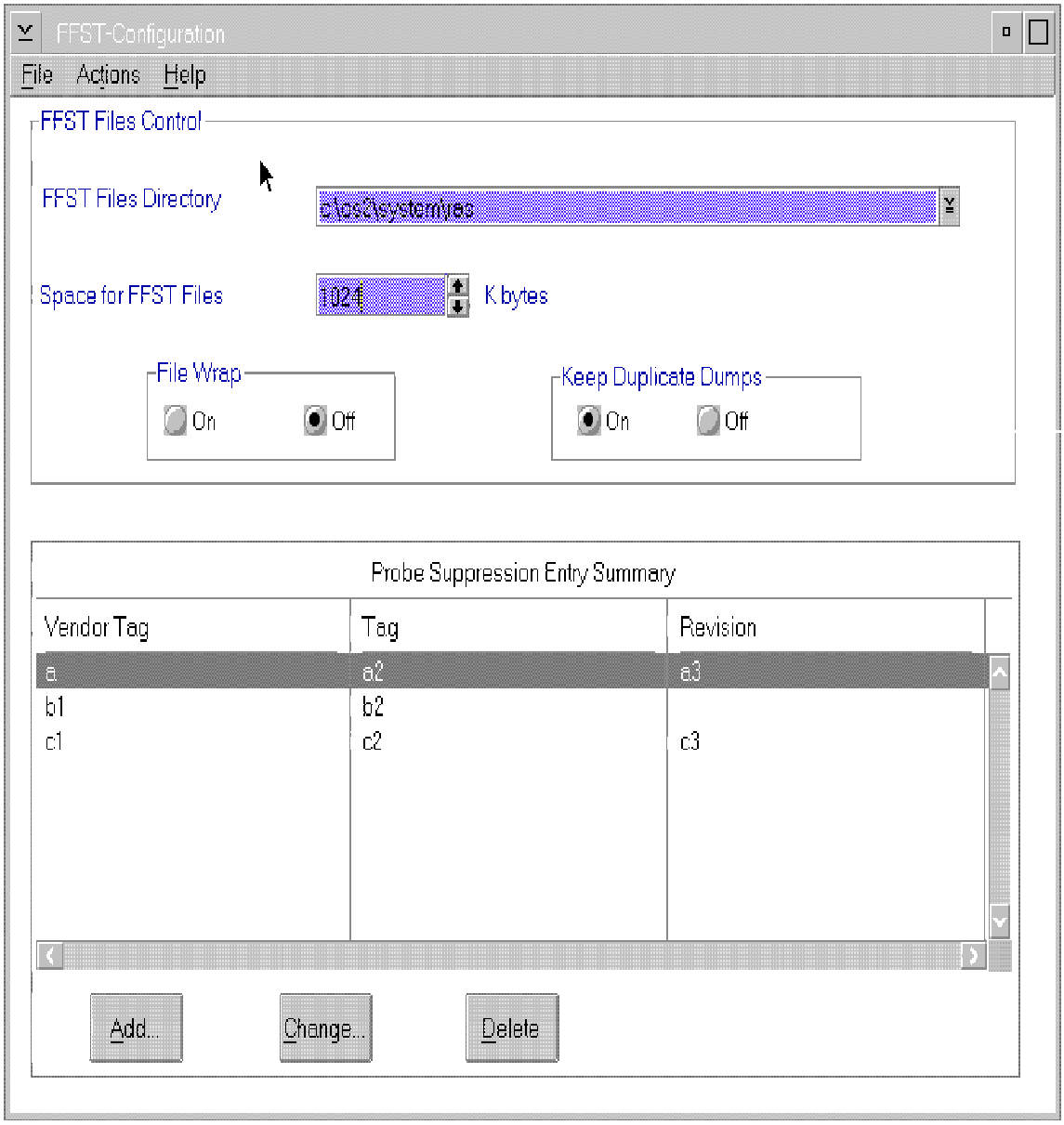
- Location of dumps, space for dumps, and so on
- Suppression of calls to groups of calls to FFSTProbe

- Information that is required to control FFST outputs (Probe Control Table Configuration)

The system saves your updated configuration information. FFST can dynamically start using the new options when you are finished with the utility.

You can use the FFSTCONF command or select the FFST Setup icon in the Problem Determination Tools folder to access the FFST Setup window (FFST Setup Window).

FFST Setup Window



You use the FFST Setup window to maintain information dealing with FFST files. The information specifies:

- where the files reside
- total maximum size for all FFST files
- whether the file should wrap when the space is full
- how to control duplicate dumps
- probe suppression for a product.

The options on the FFST Setup window are:

- **File**, which uses the Printer Setup, Print, and Exit functions.
- **Actions**, which has two options:
    - **Dumps** takes you to the FFST Dump Files Summary window, where you can select a dump to view using the PM Dump Facility dump formatter.
    - **PCT** takes you to the PCT Summary window, where you can view or edit individual calls to FFSTProbe (see Probe Control Table (PCT) User Interface).

The following actions are descriptions of the fields on the FFST Setup window. FFST dump management uses the fields to control the FFST files by defining the storage size and characteristics.

FFST Files Directory	This field specifies where to keep files that FFST creates. You can change the displayed pathname or select a new pathname. To select a new pathname, you can enter the new path name in the entry field or click on the list box selector to bring up a list of pathnames and drives. Then click on a new pathname or drive. To move up in the file tree, click on the double dot (..) entry. The default pathname is <boot-drive>\OS2\SYSTEM\RAS.
Space for FFST Files	Use this field to specify the disk storage space for FFST files. Note that the size is in kilobytes (KB). The default size is 1024 (1 MB). The number of files the system keeps depends on each file size, and on the total space that is available for FFST files. If an FFST dump exceeds the specified size, the system stores only part of the dump. The dump file information contains the reason for the partial dump and the stored parts of the dump. You will see this information when you display the dump file.
File Wrap	Select the ON value to overwrite old dumps when the specified dump file is full. Note: if this option is ON, it will not overwrite dump files that are saved by the **Save Output** choice from the PM Dump Facility dump formatter **Files** menu.
Keep Duplicate Dumps	When this option is ON, the system keeps each dump that the call to FFSTProbe creates, even when the system previously stored a dump. If this option is not used (OFF), the system requires FFST to check all the dump files. FFST checks to ensure that a dump file does not exist for this call to FFSTProbe. When you use this option, processing is slower than normal processing. Use this option to control runaway probes that create unneeded dumps that take up disk space. Even though the snapshot trace files are separate physical files, they are part of the FFST dump file so this option also affects them. This option works in conjunction with file wrap option.

Probe-Suppression Entry Summary Area

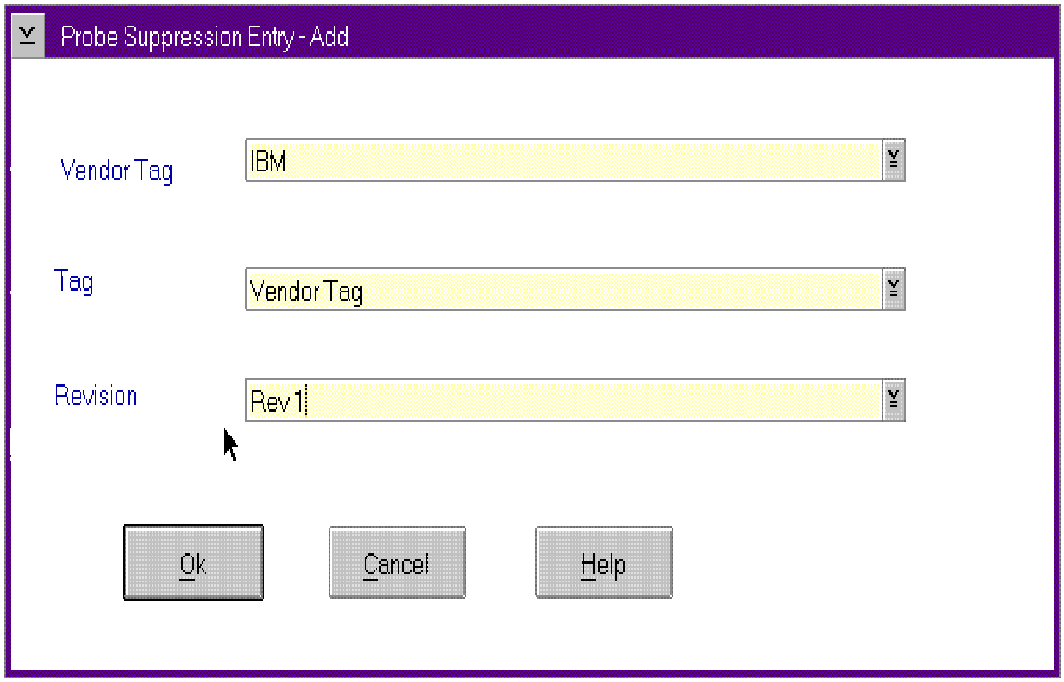
This area displays the DMI triplet of each product for which calls to FFSTProbe are to be suppressed. To start suppressing calls for your product, click on the Add push button at the bottom of the window. The Probe Suppression Entry window appears so that you can specify your product information.

Probe Suppression Entry Window (Add or Change Functions)

The Probe Suppression Entry window (Probe Suppression Entry - Add Window) appears when you select the Add or Change push buttons from the FFST Setup window. This window uses the DMI triplet for your product to control an entire group of calls to FFSTProbe within your product. Specifically, FFST uses the following DMI information:

- Vendor Tag
- Tag
- Revision

Probe Suppression Entry - Add Window



The differences among the Add, Change, and Delete screens are:

Add	Title bar reads "FFST - Probe Suppression - Add", and all fields are empty. You can add required information.
Change	Title bar reads "FFST - Probe Suppression - Change", and all fields contain information about the selected entry. You can change required information. You can click the list box selector that is located on the right side of each entry area or enter the information directly. When you click on the list box selector, all values currently stored in the PCT will be displayed. You can select the desired item from the list.
Delete	Used to delete a group of probes from Probe Suppression. This will enable information logging when calling FFSTProbe for these probes. The system will request a confirmation that you want to delete suppression.

Probe Control Table

When your code calls FFSTProbe, FFST generates the requested outputs such as dumps or log entries. These outputs can be dynamically controlled by defining an entry in the Probe Control Table (PCT). Options that are specified in a PCT entry override all other options. You can use PCT entries to do the following:

- Control the individual calls to FFSTProbe by enabling or disabling them
- Specify whether FFST and System Dump information is to be collected
- Specify whether system process information is to be collected
- Specify whether system environment information is to be collected
- Dynamically request additional system information that is not requested in the call to FFSTProbe.

Each PCT entry consists of five identifiers:

- DMI_vendor_tag
- DMI_tag
- DMI_revision
- Module identifier name
- Probe ID

The entries in the PCT describe what is to be done on the call to FFSTProbe:

- You cannot use duplicate entries in PCT
- You cannot use wildcards for identifiers

The PCT mechanism looks for specific entries that match all five identifiers. If the system cannot find a match for the entry, it ignores the probe control entry. The PCT entry makes the final decision about what to do. The entry can overwrite parameters that are specified in the configuration and change the actual call to FFSTProbe.

FFST provides no default PCT entries.

Note: It is important to understand that the Probe Control Table is used to control **individual** calls to FFSTProbe and that Probe Suppression is used for an **entire group** of calls within a product.

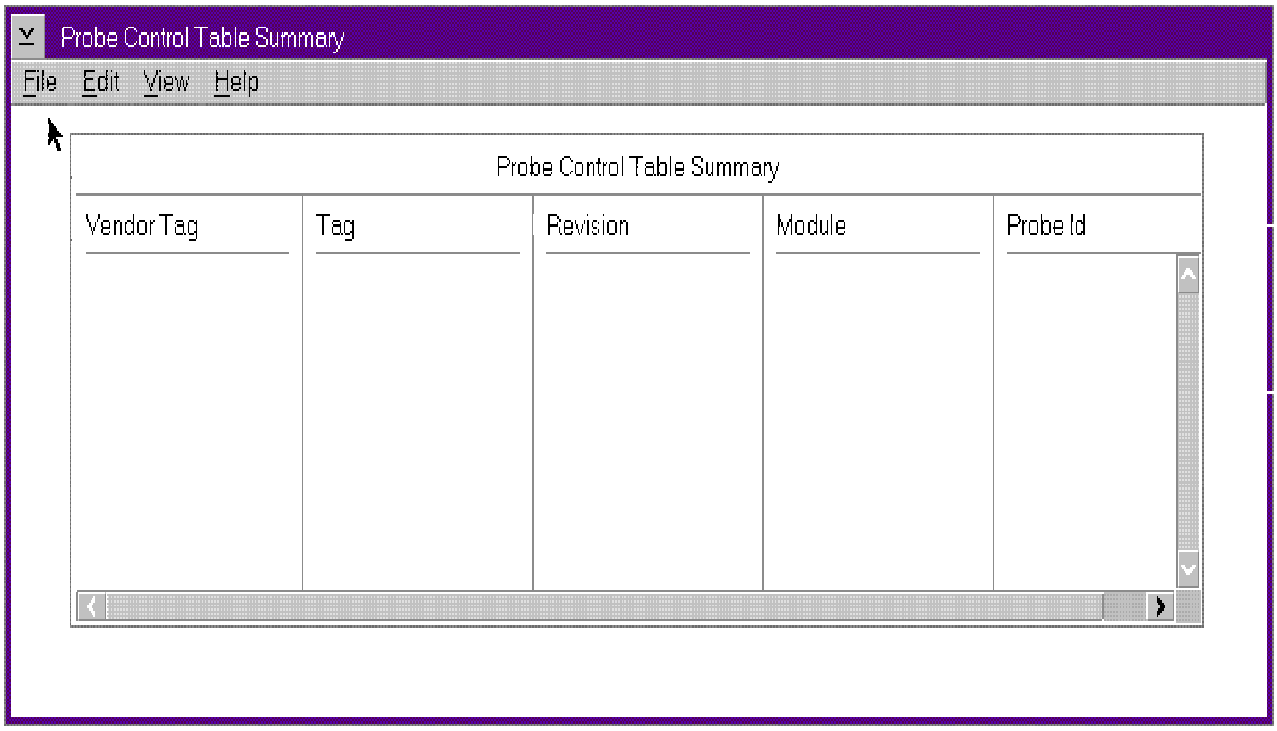
Probe Control Table (PCT) User Interface

Use the Probe Control Table to control individual calls to the FFSTProbe function.

Access the Probe Control Table by entering the FFSTCONF command, or by clicking on the FFST Setup icon in the Problem Determination Tools folder. When the FFST Setup window appears, select the **Actions** option and choose PCT.

You can use the following window to view all the defined PCT entries.

Probe Control Table Summary Window



After you select an entry on the window, use the following action-bar items.

- **File** uses the Printer Setup, Print, and Exit functions.
- **View** displays the details of the entry.
- **Edit** lets you add an entry; change details of an entry; or delete an entry.

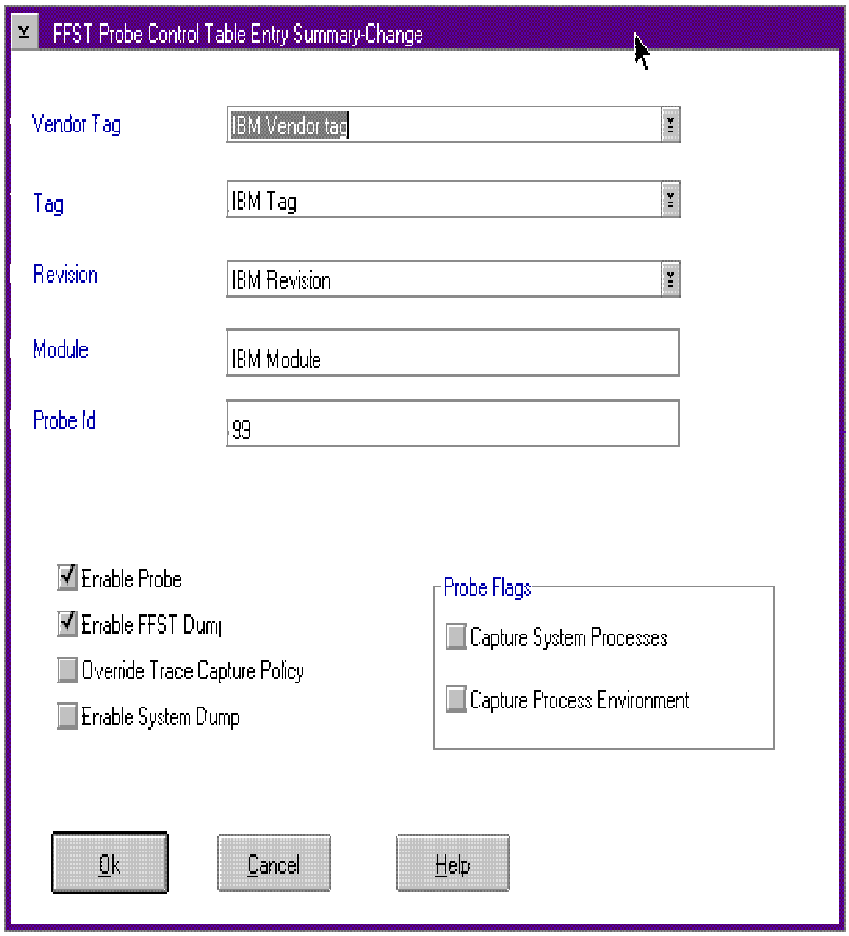
Probe Control Table (PCT) Entry Add or Change Summary Window

Use the Probe Control Table Entry Summary window (FFST Probe Control Table Entry Summary Window) to either add or change individual calls to the FFSTProbe function in a product.

Use the options on the Probe Control Table window to capture information that was not requested in the original call to the FFSTProbe function. Use the PCT options to specify system process information (PSTAT) or process environment information not in the parameters of the original call. You can also capture trace information, capture a system dump, and capture a process dump through the PCT for a specific FFSTProbe call. The system uses PCT values if the parameters on the FFSTProbe call and PCT are different.

For the PCT values to be used, the **Enable Probe** box must be "checked." The **Enable FFST Dump** box can be "checked" to have a FFST dump generated by the probe. The **Enable FFST Dump** box must be "checked" before the **Capture Trace Snapshot**, **Capture Process Dump**, **Capture System Processes**, or **Capture Process Environment** data will be captured. To capture a system dump you only need to "check" the **Enable Probe** and the **Capture System Dump** boxes.

FFST Probe Control Table Entry Summary Window



The following list defines the fields on the FFST Probe Control Entry Summary window:

Vendor Tag, Tag, and Revision	This is the VPD information for your product that is stored in DMI.
Module and Probe ID	Parameters on the call to the FFSTProbe function that identify the individual call.
Enable Probe	Used to enable or disable (turn on or turn off) the individual call to FFSTProbe.
Enable FFST Dump	Used to specify whether to save FFST Dump information for the individual call.
Capture Trace Snapshot	Used to collect trace data if your code did not specify trace in the original call to FFSTProbe.
Capture System Dump	Use this option to perform a system dump for calls to the FFSTProbe function. Note that the system restarts after storing the system dump

	information.
Capture Process Dump	Use this option to allow the system to capture process dump data. You can view the process dump data when you format the FFST dump associated with this probe ID.
Capture System Processes	Used to capture information about all the processes and threads that are running on the system. This information is similar to the PSTAT information that is viewed by using the PSTAT command. For more information about the PSTAT command, refer to the *OS/2 Warp Version 4 Command Reference* .
Capture Process Environment	Used to capture the values of environment variables for the process.

You access this window by using the following:

- From the PCT summary window, select an entry and click on either the **Details** or **Change** push button.
    - **Details** - Displays the details about the selected entry. You can only view the information.
    - **Change** - Displays the selected entry and allows information to be changed.
- From the PCT summary window, select **Add**, or **Change** from the **Edit** options.

  All fields are empty, and you add information as required.
- From the SYSLOG utility, use the **Tools** menu-bar option from any details window and select the **Modify Entry Collection** option.

  All fields contain information about the selected entry. You can change the information. If an entry is not in the PCT file, the system displays the default settings.

You cannot use wild card characters for Vendor Tag, Tag, Revision, module name, or probe ID. When you add a new entry, the system checks the PCT for duplicate entries.

Viewing and Analyzing Error Log Entries

This chapter assumes that you have read the information and are familiar with the defined terms in Guide to Instrumenting Your Code.

This chapter describes error logging and the error log formatter that is used to format, display, and analyze error information.

When your program encounters an error, the FFSTProbe API records information about the error in the system error log.

This chapter refers to the following types of error logs:

- **Active** error log: The name of the log that is currently used. When you suspend logging, the system will not write entries to the active error log.
- **Default** error log: The system uses this log after you power on the system. The system assumes this log until you specify another log to work with.
- **Current** error log: The error log that a user is currently working with.

What is an Error Record?

An *error record* is created by the system when an error in a system or application program triggers a probe in that program. Error records contain detailed information to help you diagnose the error. Error records are also called *DET1* records. Records created by a back level logging system are called *DET4* records.

What is a Control Record?

A *control record* is created by the system when you make changes to the way errors are logged. For example, when you suspend error logging or direct error logging to a new file, the system records that change in a control record. Control records are also called *DET2* records. Control records are new for FFST technology and are not available in records created by a back level logging system.

Controlling Error Logs by Using the SYSLOG Command

Use the SYSLOG command and its various parameters to access the SYSLOG utility. If you do not specify parameters when you use the command, the system loads the active system error log and displays the SYSLOG Summary window (see Working with the Error Log and Controlling Error Logging). If the system cannot find the file name you requested, SYSLOG displays a message.

If logging is not active, an error message will be shown stating

```
"The OS/2 Logging Facility device driver LOG$ is not
 loaded.  To activate logging enter DEVICE=\OS2\LOG.SYS
 and RUN=\OS2\SYSTEM\LOGDAEM.EXE in config.sys and reboot.
```

The available parameters of the SYSLOG command are as follows:

Parameter	Action
/V:<error log pathname>	Use this parameter with the error log pathname to access the error log. If you specify no pathname, the system accesses the active error log.
/S:<error log pathname>	Use this parameter to *suspend* error logging. Note that when you use this parameter, the system writes an error log entry to the current active error log that indicates that you suspended logging.
/R:<error log pathname>	Use this parameter to *resume* error logging to the suspended file. If the system cannot find the pathname, the system does not resume logging. If the system finds the pathname but cannot find the error log file, the system creates the error log file and resumes logging. Note that when you use this parameter the system writes an error log entry to the current active error log that indicates that you resumed logging.
/P:<error log pathname>	Use this parameter to *redirect* error logging to another error log. Note that when you use this parameter the system writes an error log entry to the current active error log that indicates you redirected logging. The error log entry contains the pathname of the error log file that logging was directed from and the pathname of the error log file that logging was directed to.
/W:xx	Use this parameter along with the /P parameter and the error log pathname to change the *maximum size* (xx) of the specified error log. The default size of the error log is 64 KB.

Viewing Error Log Contents by Using SYSLOG

SYSLOG is also a function you use to view the contents of an error log entry. This function resides in the Problem Determination Tools folder.

SYSLOG supports formatting and viewing error log entries created either from the DosProbe or FFSTProbe APIs. Entries created by DosProbe will not have as much data

You can start SYSLOG by any of the following methods:

- Clicking on the SYSLOG icon that is located in the *Problem Determination Tools* folder of the *OS2 System* folder.
- Using the SYSLOG command without any parameters. The *Problem Determination Users Guide* in the *Problem Determination Tools* folder describes the SYSLOG command.

- Dragging an error log file and dropping it onto the SYSLOG icon.

If you drop multiple error log files onto the SYSLOG icon, the system starts a separate session of the formatter for each file.

You can select another error log file to view by selecting the **Open...** choice of the **File** pull-down option within SYSLOG. The active file appears as the default. You can change the file name to any other error log.

You can select which error log entry to view by double clicking on an error log entry. SYSLOG windows contain the following system error log information:

- A summary panel containing an entry for each error log record in a given error log file with limited information on each.
- One or more panels containing detailed information on specific error log entries.
- A browseable pop-up window containing the header information for the error log being viewed.
- An entry into the PM Dump Facility dump formatter tool using parameters the error log record contains.
- An entry into the trace formatter tool using the parameters that are contained in the error log record.
- An entry into the Modify Enter Collection tool used to change data-collection parameters.

In addition to its display capabilities, SYSLOG also provides controls to change the system parameters that pertain to logging. Use the controls to do the following:

- activate logging
- suspend active logging
- redirect logging
- increase the maximum size of the error log file.

The following information describes the SYSLOG Summary window and explains how you use SYSLOG to view, change, and control the error log and error log contents.

Working with the Error Log and Controlling Error Logging

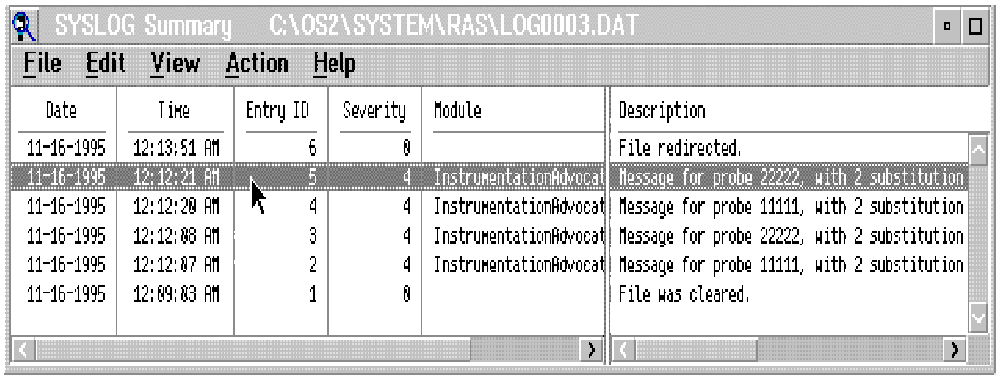
When the system opens the error log, SYSLOG displays a Summary window with the following information for each error log entry:

- the date and time the system detected the error
- entry ID number
- severity of the error
- module name
- error description.

You use the Summary window to work with error logs, error log entries, and control the error-logging function.

To see all the details for each error log entry, use the horizontal slider bar. You can select Help on the menu bar for descriptions of the fields and options on the Summary window.

SYSLOG Summary Window



The Summary window is a snapshot of the error log. It allows selection of single or multiple entries.

Be aware that, when you use the *active* log, the log may change during the session if the system records another error. The error log details for a selected entry may no longer be available from the error log file if the error log wraps. When wrapping occurs, the system adds new entries at the top of the error log and erases the oldest entries. The system erases only enough old information to make room for the new for the new information. If you select an erased error entry, the system displays an error message.

The following information contains a description of the options on the Summary window.

File Choices

The **File** choices that are accessed from the SYSLOG Summary window are the standard OS/2 choices: **Open..., Printer Setup...**, and **Print**.

When you select the **Open...** choice, the system displays the Open an Error Log File window.

The system uses the default error log named LOG001.DAT when opening the error log for the first time. LOG001.DAT remains the summary log until you redirect logging to another log (see Redirect Logging).

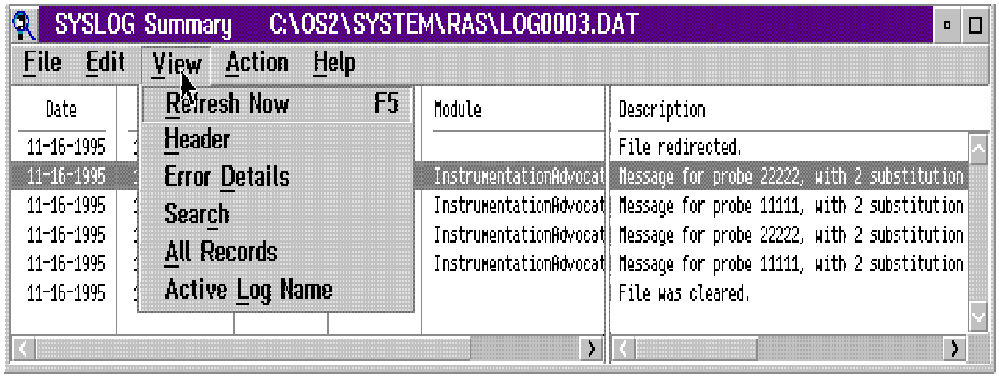
Edit Choices

The **Edit** choices that you access from the SYSLOG Summary window are the standard OS/2 choices: **Copy, Select All,** and **Deselect All.**

View Choices

The **View** choices from the SYSLOG Summary window are **Refresh Now, Header, Error Details, Search, All Records,** and **Active File Name.** The description of the **View** choices follow.

View Menu-Bar Choices on SYSLOG Summary Window



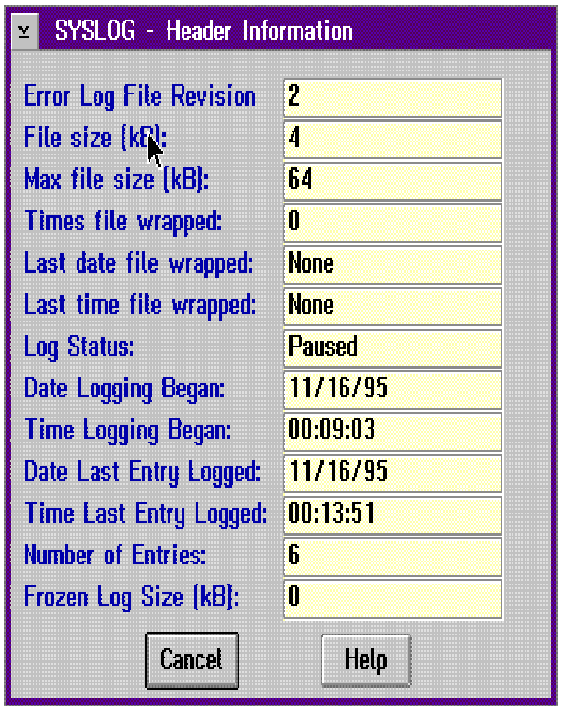
Refresh Now

The **Refresh Now** choice updates the Summary window with any changes that occurred in the error log after you initially displayed the log. If the error log being viewed is the system's active log and logging is not suspended, the contents of the error log could change during the SYSLOG session. Values that are displayed on the Search window have no effect on the records displayed in the Summary window when you use this option. The system ignores prior search operations.

Header

The **Header** choice displays the header information of the error log entry being viewed.

SYSLOG Header Information Window



Error Details

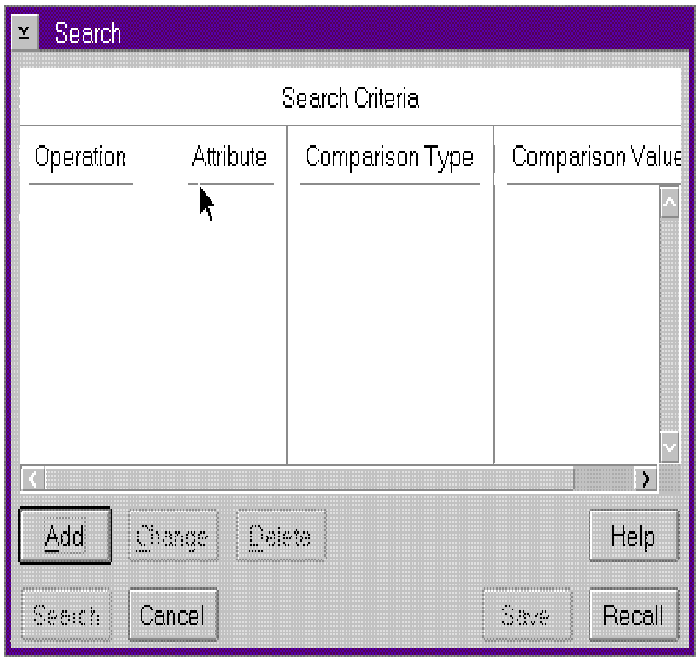
The **Error Details** choice displays a SYSLOG Details window that can be used to access dump and trace data and to change error-entry collections. This choice associates various tools and appears later in this chapter under the heading Displaying Error Log Entry Data.

Search Choice

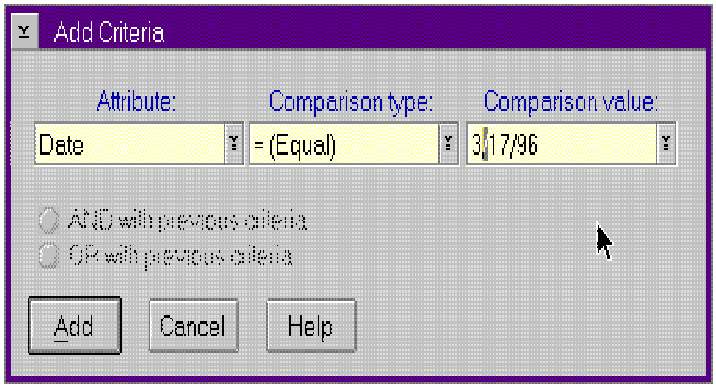
The **Search** choice uses your search values to select the log entries that are displayed on the Summary window. The **Search** choice has three associated windows: Search window, Add Criteria dialog, and Change Criteria dialog.

The Search window is a dialog window with fields for you to select, create, change, and delete search criteria. You can use search values alone (by specifying OR) or combined (by specifying AND) to provide either specific or general filtering.

Search Window



Add Criteria Dialog



Use the Add Criteria dialog to add criteria to the search for log entries. Use the three entry fields (Attribute, Comparison Type, and Comparison Value) to construct the criteria. Each entry field has a pull-down list box that contains valid values for the field. You can select an item from the list box, or enter text in the field.

The valid values for the Attribute field are:

- Date
- Time
- Entry ID
- Record type
- Severity
- Directory name
- Module
- Probe
- DMI vendor tag
- DMI tag
- Machine type
- Serial number
- User data

The Comparison Type values are the standard OS/2 values. Note that some values may not be valid with certain Attributes. For example, the Greater Than type is not valid with the Directory Name attribute. In this case, the system displays an error message box.

The Comparison Value may not always have a pull-down list box, depending on the Attribute value being selected.

Use the Change Criteria dialog to change the highlighted criteria on the Search window. It is similar to the Add Criteria dialog.

All Records

The **All Records** choice restores the contents of the Summary window to the previous search values.

Active Log Name

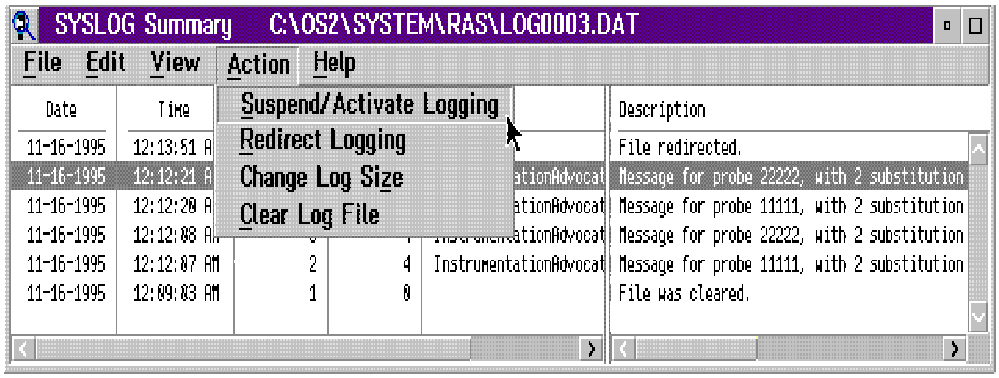
The **Active Log Name** choice displays the active log name, which is the error log that is currently being used for the logging of errors. This may or may not be the log you are currently viewing.

Action Choices

The **Action** choices from the SYSLOG Summary window are **Suspend/Activate Logging, Redirect Logging, Change Log Size,** and **Clear Log File.**

All the **Action** menu choices pertain to the actual collection of system error information. For each action you take, the system records the action in the appropriate error log.

Action Menu-Bar Choices on the SYSLOG Summary Window



Suspend/Activate Logging

The **Suspend/Activate Logging** choice causes the system to either activate or suspend the logging of errors to the current log. If logging is active when you select this option, the system suspends logging. When logging is inactive, selecting this option reactivates logging. After suspending logging, you can delete the log file and reactivate logging. The system creates a new log file with the same name as the deleted file and resumes logging to the new file.

The system writes a record entry to the current error log file for each change in logging status.

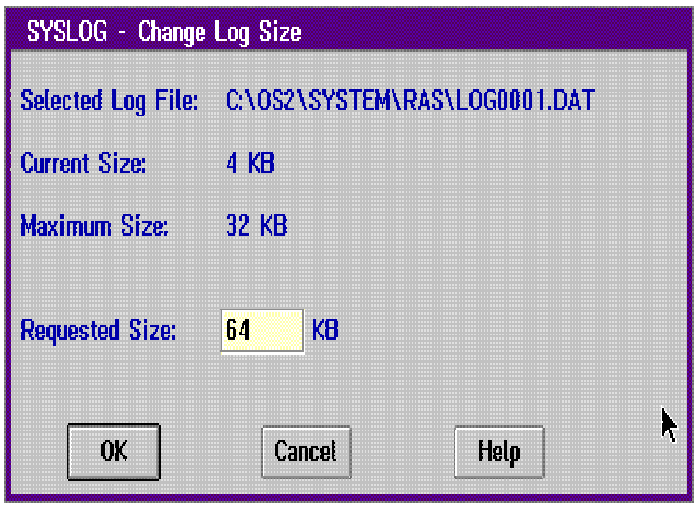
Redirect Logging

The **Redirect Logging** choice displays a standard file dialog for you to choose a different destination for error logging. You can select an

existing log file or a new log file. If you direct logging to a log file that does not exist, the system creates the file and directs logging to the new file. The system writes an entry to the previously active error log file to indicate the change in logging status.

Change Log Size

Change Log Size Window

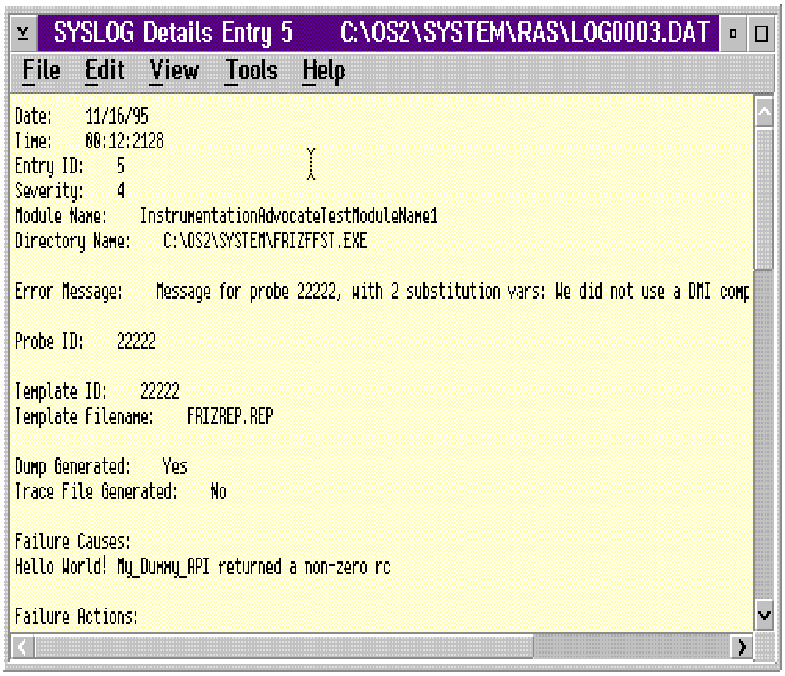


The **Change Log Size** choice allows users to change the maximum size of the log file that the system allocates. You can either increase or decrease the size of the log. You cannot make the size smaller than the current size. Selecting this choice displays the path name of the system's current log file as well as any other log files that are known to the system. The system also displays the current maximum size and an input field allowing the maximum size to be changed. When complete, a message box informs the user of the new status. This choice does not change the maximum size of log files other than the selected log. The system writes a log entry to the error log file you specified to indicate the change in size.

Displaying Error Log Entry Data

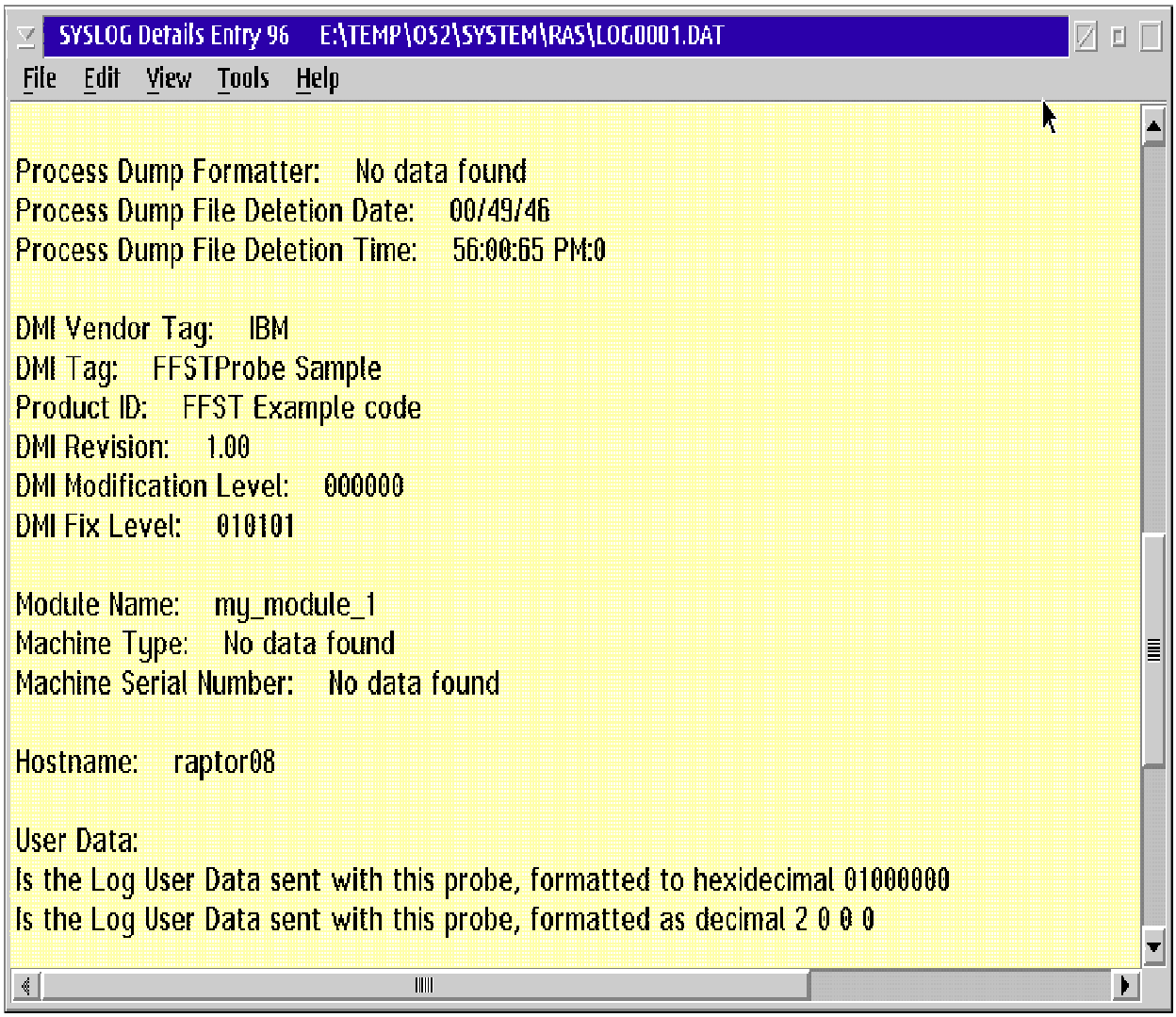
Select the **Error Details** choice from the **View** menu on the SYSLOG Summary window to display a Details screen for the selected log entry. The Details screen below shows DET1 detailed information about a log entry created using the *current* FFST technology. The **File** and **Edit** menu-bar choices are standard OS/2 choices.

SYSLOG DET1 Record Details Window



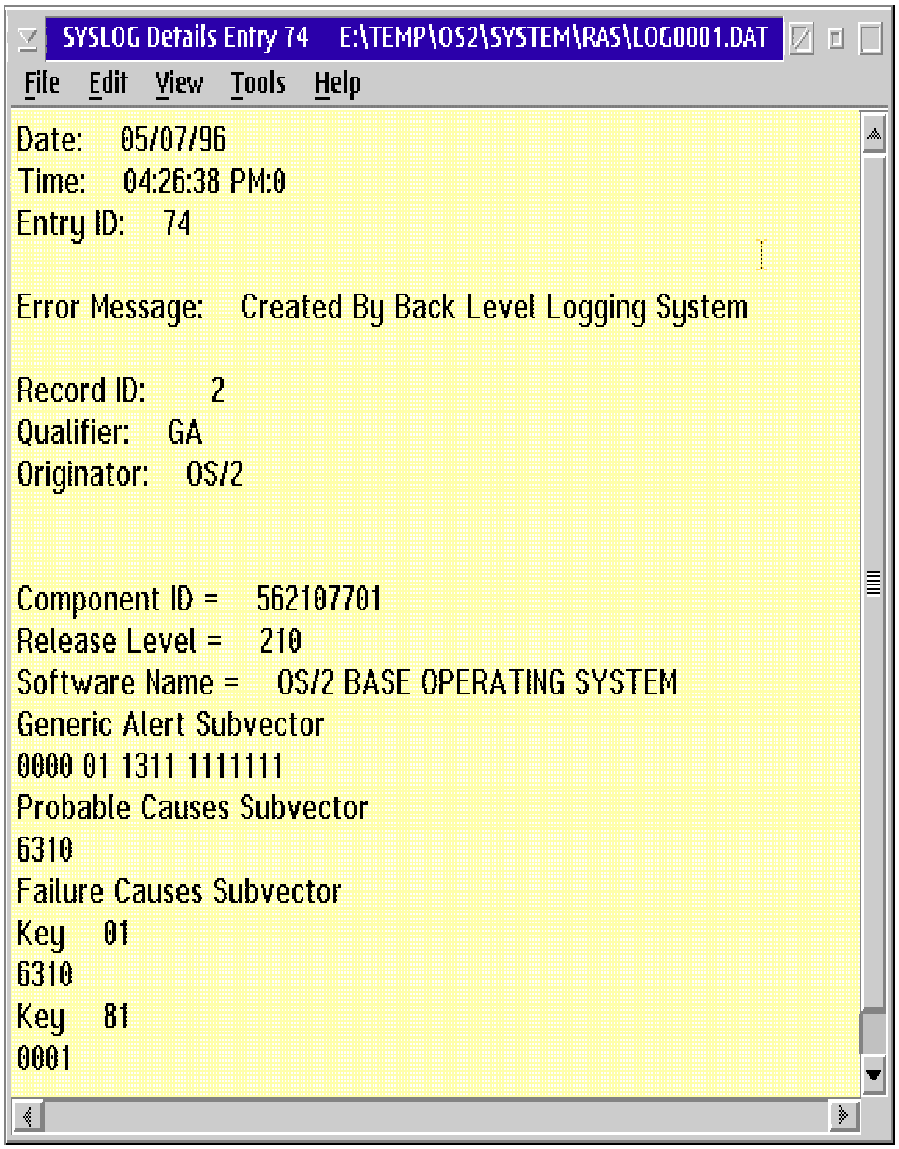
The following screen shows an example of User Data that was requested in the parameters when the FFSTProbe API was called.

SYSLOG DET1 Record User Data Info



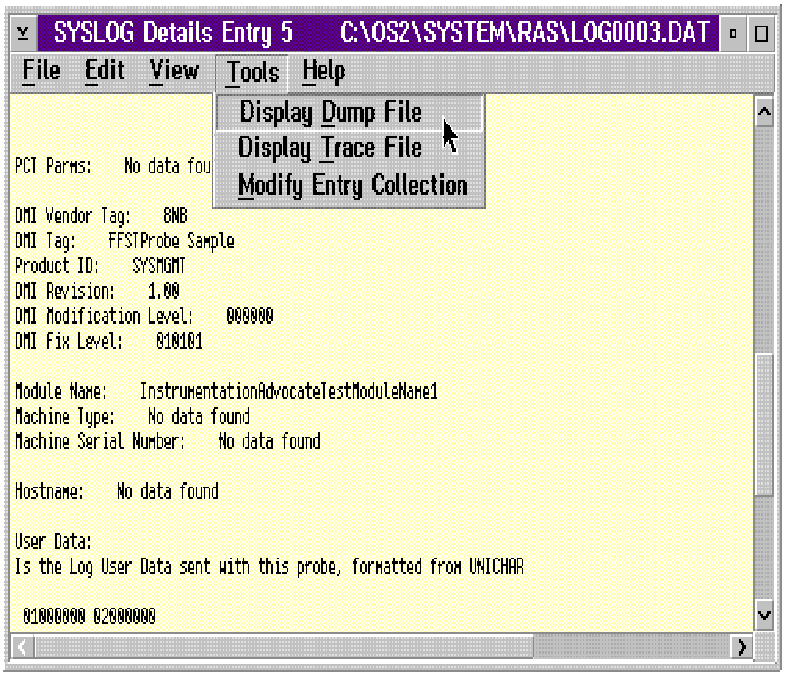
The Details screen shown below is an example of a DET4 error record detailed information created using *back level* FFST technology. The formats of the data may vary.

SYSLOG DET4 Record Details Window



Select the **Tools** menu-bar choice from the Details window that is shown in Error Details to access dump and trace data and change entry collections.

Tools Menu-Bar Choices on the SYSLOG Details Window



The **Display Dump File** choice starts the PM Dump Facility dump formatter by using the parameters that are contained in the log record being displayed. Capturing and Saving Failure-Related Information through Dumps contains more information about dumps.

The **Display Trace File** choice starts the trace formatter by using the parameters that are contained in the log record being displayed. Analyzing Performance and Debugging Problems Using Trace contains more information about trace.

The **Modify Entry Collection** choice starts the PCT (probe control table) function of FFST. You can use the PCT tool to change options that are associated with the call to FFSTProbe that generated this log entry. For more information about the PCT function, see Probe Control Table.

Accessing Error-Log Information through Functions

You can also work with error log information through functions that provide an open interface by allowing OS/2 applications to access error log entries. These functions are:

- LogOpenFile - used to open a log file for subsequent reading.
- LogReadEntry - used to read entries from the log file. The call supports both a direct access mechanism and a log-file search mechanism.
- LogFormatEntry - used to obtain a set of ASCII or UniCode strings that you can display after formatting the log entry.
- LogCloseFile - used to close a Log File that a LogOpenFile call opened.

As described in Problem Determination APIs, these APIs can be used with either ASCII or UniCode data.

Event Notification

Each time your code calls the FFSTProbe function, the system creates an error log entry. This action is known as an *event*. You can request to be notified when the system records errors. The system notifies you through a process that is known as *event notification*.

You use an event-notification filter, a flexible data structure, to specify the type of events for notification.

The functions that are associated with event notification are:

- LogOpenEventNotification

  This function enables your product to register with the logging service so that the system notifies you when it creates specific records. You specify which log records you would like to be notified about by providing filtering information. If you do not provide filter information, the system notifies you of every entry.

- LogChangeEventFilter

  This function enables you to change the event-notification filter for a registered product. In addition to changing the filter, you can specify current event-notification filter entries for the system to delete before the filter change takes effect.

- LogWaitEvent

  After the LogOpenEventNotification function registers your product with the logging service, use the LogWaitEvent function to request notification. The system notifies you of the next error log entry that meets the registration values. The system returns the log entry into a buffer area your product specified.

- LogCloseEventNotification

  This function enables you to close event-notification requests and remove product registration with the logging service.

For more information about these functions, refer to Problem Determination APIs.

Remote Error Reporting

A local system uses remote error reporting to notify a remote managing system that an error has occurred. You must enable remote error reporting by added the **/r** option on the SMSTART statement in the CONFIG.SYS file.

The remote error-reporting application on the local system monitors the error log. When the system records an error, the remote error-reporting application converts the error into a Desktop Management Interface (DMI) indication and notifies the SystemView Agent of the error. The SystemView Agent converts the indication into an SNMP trap and sends the trap to a remote error-managing system. The indications that the remote error-reporting application creates are also available to local system applications through the DMI Management Interface.

As the system records errors through use of the FFSTProbe function, the remote error logging application uses the logging event-notification functions to receive the error log entries. The remote error logging application translates the entries into DMI indications. The system sends DMI indications to the DMI service-layer program. The DMI service layer forwards the indications to DMI management application programs. One of these applications, the DMI Subagent, is a SystemView program that converts the DMI indications into SNMP (Simple Network Management Protocol) traps. After conversion, the system sends the traps to an SNMP manager program that resides on a remote system. The SNMP manager receives the traps and displays error log fields so that a network administrator can read the traps to determine appropriate action.

The current implementation translates a Unicode-format error log entry into an ISO8859-1 format DMI indication. The system changes any Unicode characters that cannot be displayed in ISO8859-1 to a period (.).

Building and Sending an Indication

When the remote error reporting application receives an entry from the error log, it translates the entry into a DMI indication. An error log entry contains multiple fields of information that describe the error being logged. The remote error reporting application translates several of the fields into DMI attributes. The attributes match the remote error-reporting System Management Information Format (MIF) the system installed in the DMI MIF database.

The DMI indication sends only a portion of the error log entry. The indication contains a total of 17 attributes from three error groups. The IBM Event Indication group contains general attributes that identify the error. The FFST Error and OS/2 Software groups contain fields that provide specific information. For more detail of these error groups, view the remote error-reporting MIF (REMOTERR.MIF).

Because the lengths of some attributes can be as long as 508 bytes. The indication, once converted into an SNMP trap, could exceed the maximum length of 4096 bytes. Normally indications do not exceed the maximum size.

The system sends DMI indications for Only error records with severity of the following levels: critical error, major error, and minor error.

For additional information that is related to remote error reporting, refer to *SystemView Agent User's Guide* and the *SystemView Agent Programmer's Guide* .

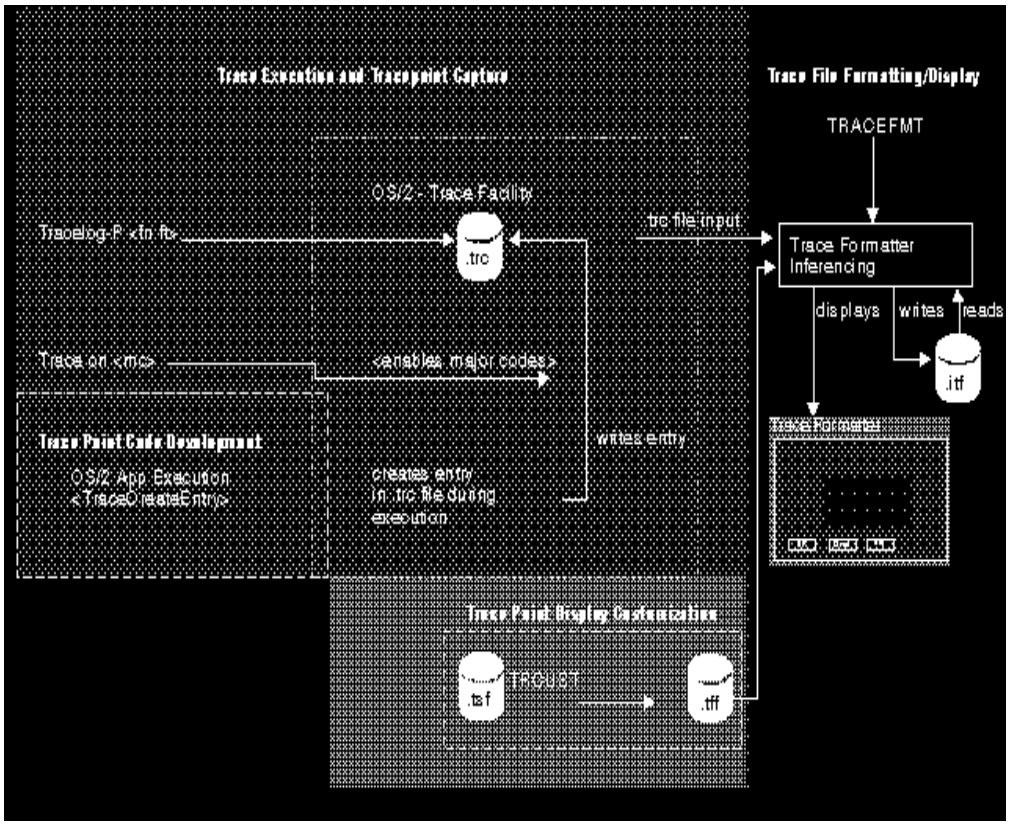
Analyzing Performance and Debugging Problems Using Trace

This chapter describes trace and the benefits of the various types of trace points. The chapter contains an overview of instrumenting (setting up) your code to use trace and explains how to display and analyze trace information.

This chapter also describes the following programs that are used to analyze trace information:

- Trace Formatter (TRACEFMT)
- Trace Commands (TRACE and TRACEBUF)
- Trace Customizer (TRCUST)
- Trace Capture (TRACEGET)

Trace Flow



Turning on Trace Points Using the TRACE Command

Use the TRACE command to control (turn on and off) trace points in your product. When you turn on a trace point, the system considers the trace point to be active and stores information for trace point in the trace file.

Refer to *OS/2 Warp Version 4 Command Reference* for information about using the TRACE command and its parameters.

The following two statements need to be added to your CONFIG.SYS file before you use the TRACE command:

TRACEBUF=x *x* is used to set the trace buffer size. Valid values are from 1 to 63 kilobytes (KB). The default

	size of the buffer is 4 KB.
TRACE=ì ON | OFF	Used to turn trace on or off.

If neither statement (TRACEBUF nor TRACE) is included in the CONFIG.SYS file on your system, trace is disabled.

Trace Scenario

Use the TRACE command to store trace data in the trace buffer. The following list of commands and descriptions define what happens when you use the TRACE command.

Command Entered	**Description of Function**
TRACE=ON	Statement added to CONFIG.SYS
TRACEBUF=4	Statement added to CONFIG.SYS
TRCUST TEST.TSF	Creates trace information for both formatting output text and for controlling dynamic trace. For more information about the TRCUST command and its parameters, refer to the TRCUST document in the Toolkit.
TRACE ON 240	Turn trace ON and specify the *major code* for the trace points to be traced.
TRACE ON TEST.TDF	Modifies the .DLL file and starts dynamic trace points in the .DLL file specified in the TEST.TSF file.
Run your program	Perform normal system operations
TEST	
TRACE OFF	Use the TRACE command and OFF parameter to end tracing
TRACEFMT	Use the TRACEFMT command to format event trace buffer data.

For information about the TRACE and TRACEBUF commands, refer to the *OS/2 Warp Version 4 Command Reference* .

Saving Trace Data

You save trace point entries that are created within the system trace buffer in the following ways:

- Use TRACE ON to turn on all static trace points
- Use TRACE ON 220 to turn on trace points for major code 220
- Use TRACE ON PRE to turn on the trace points for the PRE grouping. There are several other groupings available.
- Use TRACE ON TEST.TDF to patch the .DLL file named in the TEST.TSF file. TRCUST uses the TEST.TSF file to create the TEST.TDF file. After the .DLL is patched, the system will save trace points when the .DLL is loaded into the system and run.

There are several other combinations of using the TRACE command. Refer to the *OS/2 Warp Version 4 Command Reference* for more information.

Formatting Trace Data

The system saves collected trace points in the system trace buffer. This buffer can be displayed by using the trace formatter. This section includes descriptions of the execution methods, user interface, supported files, and the formatting of data.

As seen in the example in Trace Format, you can display two types of information when you use the trace formatter. Summary data at the top of the file includes information about the file, trace buffer, and possibly vital product data (VPD). This information changes with each trace file.

The second type of data are the actual trace point entries. Each entry includes the following:

- event number
- timestamp
- major and minor codes
- process ID information
- any optional data saved by the system.

Starting the Trace Formatter

You start the trace formatter by using any of the following methods:

- On an OS/2 command line, type **TRACEFMT** (followed by an optional trace file name).
- Double-click on the trace formatter program icon.
- Drag and drop the icon for a trace data file onto the trace format program icon.
- Double-click on the icon for a trace data file that is associated with the trace format program.
- Start the trace formatter option from the SYSLOG program or the PMDF program.

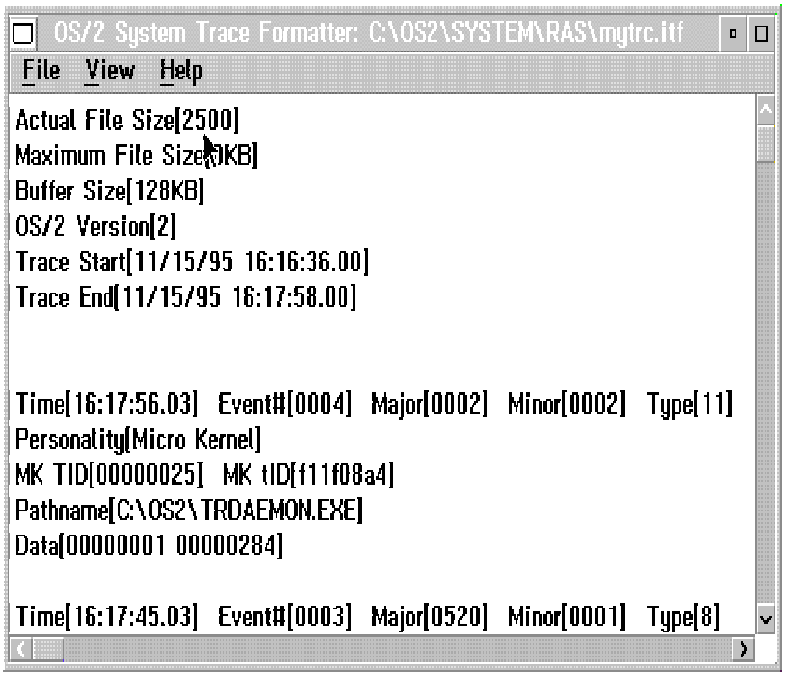
Using the Trace Formatter to View, Change, and Analyze Trace Data

This section describes the user interface that the trace formatter uses.

Menu-Bar Choices

The pull-downs available within the trace formatter are:

Trace Format



File	Use this pull-down to select a trace file to viewed and what to do to the contents:	
	Open	Used to select a .TRC file to view;
	Save unformatted	Used to save unformatted trace data to a file usually with the file extension of .itf. The .itf file can be reloaded into the trace formatter.
	Save formatted	Used to save the contents to a text file (with the extension .FTF for Formatted Trace File). This data can be viewed with an editor such as the system editor.
	Print Setup	Used to select the output destination of formatted data when you select the **Print** option. Output can go to either a printer or a file; **check** the corresponding box to choose the destination. Select a printer name from the drop-down choices or type a file name directly into the field.
	Print	Used to send the currently formatted data to the destination selected in the **Printer Setup** option.
	Recapture buffer	Recaptures the current trace buffer and refreshes the display.
	Set .TFF search path	Set where to look for .TFF trace formatting files to use when you format trace data.
	Exit	Ends the trace formatter program.
View	Allows you to change the view of the trace file contents:	
	Select	Allows a combination of criteria to be selected. You can choose the trace contents important to viewing; this is similar to the SYSLOG Search function as shown in Search Choice. You select the following items in various combinations: • Major codes • Minor codes • Event number • Process ID • Time and date The search combination allows you to build search selections by adding, deleting, or changing conditions before you start the search. In addition, you can save the search values for reuse. The system refreshes the window with the selected contents.

	Display all events	Cancels the previous **Select** or summary display and shows all contents.
	Display summary by Process ID	Displays a sorted listing by process ID
	Display summary by Major Code	Displays a sorted listing by major code
	Find	Allows you to search for any particular text in the displayed contents.
	Repeat find	Repeats the search for the string of text.
	Mark all	Marks all the text being displayed so you can then copy the text to the clipboard.
	Copy	Copy the selected text contents to the clipboard. You cannot paste from the clipboard to the trace formatter.
	Fonts	Select the font used to display formatted data.
Help		
	Using help	General help on using trace formatter
	General help	General help on using help
	Keys help	General help on using trace formatter keys
	Help index	Index of help key words
	Product Information	Copyright and program information

TRACEGET Command

When you enter the command *TRACEGET nnn.trc*, the system captures the contents of the trace buffer into the file named *nnn.trc*. The TRACEGET command is a supplement to the trace formatter. You use the command when the trace formatter cannot be used (such as when a Presentation Manager failure occurs). The file *nnn.trc* can be saved on removable media and sent to your service organization. You can also copy *nnn.trc* to another system and use the trace formatter to display the contents of the file.

Summary

This chapter has covered the user interface functions and commands that are associated with trace.

You use trace to understand data and functions both outside and inside of applications. Use this data to correct problems and resolve unwanted conditions.

The trace facility gives you a means to track, save, display, and analyze this diagnostic program data.

Capturing and Saving Failure-Related Information through Dumps

This chapter describes FFST dumps, process dumps, and system dumps. This chapter also instructs you how to set up for and enable system dumps. This chapter instructs you on how to use the Presentation Manager Dump Facility (PMDF) dump formatter to display and analyze FFST (First Failure Support Technology) dumps, process dump, and system dump information.

The FFST dump portion of this chapter assumes that you are familiar with the information and terms that are described in Guide to Instrumenting Your Code.

What Is A Dump?

A dump is a file created by the system at the time of a failure that contains a collection of system data. The collected data is analyzed by software service personnel to determine the cause of a software problem. There are three types of dumps possible for OS/2 Warp Version 4.

- FFST dump

  The system generates a FFST dump when the FFSTProbe function requests certain user and process data. The dump collects the application-oriented and program-specific information that you specify. Guide to Instrumenting Your Code covers the FFSTProbe function, the FFST dump function, and code instrumentation for this type of activity. This chapter contains information about formatting and displaying the contents of the FFST dump.

- Process dump

  A process dump contains limited information about a single process that was running at the time of the failure. The system dump contains greater detail about all processes that were running when the failure occurred. The process dump can be formatted using the PM Dump Facility dump formatter described in this chapter.

- System dump

  A system dump covers system-wide activities. When a system dump occurs, the system automatically stops and stores the contents of main memory. After the data is stored, the system automatically reboots and any operating data that is not in main memory is lost. In other words, when the operating system software encounters a serious problem, the system triggers a system dump. The dump contains the important failure-related information held in the system main memory. The system records this information in a system dump file. You use the information to debug and solve system problems. This chapter discusses the system dump file later.

Where Are Dumps Stored?

FFST dumps are stored stored in the directory specified in the FFST setup. If you do not specify where to store FFST dumps, the system will store the FFST dump file in the default directory of <boot-drive>\OS2\SYSTEM\RAS. The FFST dump file will have the extention of *DMP*.

Process dumps can be stored only in the root directory of the drive specified in the DUMPPROCESS statement in the CONFIG.SYS file (see Setting Up for a Process Dump). The file name will be *PDUMP.nnn* where *nnn* is a number that is incremented each time a process dump is generated.

System dumps are stored either in a dedicated FAT hard disk partition on your system or on a diskette. For details on setting up the dump partition refer to Setting Up for a System Dump for more information.

System dumps taken to a hard disk partition may be used directly by the PM Dump Facility dump formatter.

Any system dump stored on diskettes is compressed and needs to be decompressed to produce a single dump file. This may be done directly from PMDF by selecting the **Decompress Dump** option of the File pull-down. PMDF offers the additional facility of decompressing diskette dumps directly from diskette images (see PM Dump Facility Dump Formatter File Option).

System Dump

A System dump requires that the following steps are performed in order:

1. You set up for a system dump by adding statements to the CONFIG.SYS file and adding instrumentation to your code
2. The system performs the dump

3. You use the PM Dump Facility dump formatter to format and display dump information in a way that you can analyze it.

Setting Up for a System Dump

You must prepare your system to store system dumps before the dump occurs. You add the TRAPDUMP statement described below to the CONFIG.SYS file on your system. The TRAPDUMP statement enables your system to store dump data.

The TRAPDUMP statement controls the system dump facility of OS/2. It will enable initiation of a system dump the instant a trap (error) occurs. The TRAPDUMP statement tells the system where to store the dump information: either on formatted diskettes or on your system's disk storage. Dumping to diskette is the default. A dump is stored on disk storage if the second parameter is used shown as **X:** in the TRAPDUMP statement example below.

The disk storage device on your system must have a file allocation table (FAT) partition with the volume label of SADUMP. TRAPDUMP accesses this partition when the dump information is stored. You specify the partition with second parameter in the TRAPDUMP statement.

Note: Do not specify a partition that contains vital data. When a dump is written to the partition, it overwrites any data that is in the partition.

To enable system dumps to be performed you need to add the following statement to the CONFIG.SYS file on your system:

TRAPDUMP= ì OFF | ON | R0 ì , X:

OFF
: Specifies that the stand-alone dump process will not initiate automatically when an unrecoverable trap occurs. This is the default option. It does not prohibit the use of the Ctrl-Alt-Numlock-Numlock key sequence, Ctrl-Alt-F10-F10 key sequence, or the use of DosForceSystemDump in your code to force a system dump to be performed.

ON
: Specifies that the stand-alone dump process will be automatically initiated whenever an unrecoverable trap occurs.

R0
: Specifies that only ring zero traps will automatically initiate the system dump process.

X:
: specifies the hard-disk FAT partition to which OS2DUMP will write a stand-alone dump. The partition letter must have the colon suffix.

  Note:

  1. The partition may be specified with either ON or OFF. When specified with OFF it will allow a stand-alone dump initiated by Ctrl-Alt-Numlock-Numlock or Ctrl-Alt-F10-F10 to be written to the dump partition.
  2. The only removable media you can use for dump storage is diskettes.
  3. Only hard disk logical drives and primary partitions may be specified.
  4. The system will erase all data on the dump media (disk partition or diskettes) before writing the dump. Do not specify a disk partition or use diskettes that contain vital data.
  5. When dumping to disk storage, the system is automatically re-booted on completion of the dump if there is a REIPL=ON statement in the CONFIG.SYS file. If the dump was stored on diskettes, the system will not be re-booted automatically.

You can add the REIPL statement to CONFIG.SYS to allow the system to re-boot (re-IPL) following an error.

The syntax of the statement is as follows:

REIPL=ON | OFF

The statement has the following parameters:

ON
: This specifies the system it to be automatically re-booted following an error.

OFF
: This specifies that the system is not to be automatically re-booted following an error. The system will remain hung until manually restarted.

Starting a System Dump

You start, or trigger, system dumps in several different ways:

1. Using the keyboard sequence (Ctrl-Alt-F10-F10 or Ctrl-Alt-Numlock-Numlock)
2. Calling the DosForceSystemDump API
3. Selecting the *Enable System Dump* choice on the Probe Control Table and allowing the FFSTProbe API to generate the dump

If TRAPDUMP=ON statement is in the CONFIG.SYS file and a trap occurs, a system dump will automatically be generated.

DosForceSystemDump Function

For more information about the DosForceSystemDump function, refer to the *OS/2 Warp Version 4 CP Reference* in the Toolkit.

FFSTProbe Function

The FFSTProbe method of triggering is done by selecting the **Enable System Dump** option on the FFST Probe Control Table Entry Summary - Add window shown on page Probe Control Table (PCT) Entry Add or Change Summary Window. When your code calls FFSTProbe for a Probe ID that you specified on FFST Probe Control Table Entry Summary - Add window a system dump starts immediately. The FFSTProbe function calls the DosForceSystemDump function to perform the dump.

Process Dump

A process dump file contains very basic unformatted system and user storage data related to the process that encountered the error and does not contain any data pertaining to main memory.

A process dump requires that the following steps are performed in order:

1. You set up for a process dump by adding statements to the CONFIG.SYS file and adding instrumentation to your code
2. The system performs the dump
3. You use the PM Dump Facility dump formatter to format and display dump information in a way that you can analyze it.

Setting Up for a Process Dump

The DUMPPROCESS statement in the CONFIG.SYS file on your system allows you to activate the process dump facility. When a process dump file is created the file name takes the form of PDUMP.nnn where nnn is an index that is incremented each time a new process dump is created.

The syntax of the DUMPPROCESS statement in the CONFIG.SYS file is defined below.

DUMPPROCESS=x

x

This parameter specifies the drive letter (excluding the colon) on system where the process dump data file is stored. The file takes the name PDUMP.nnn and resides in the root directory of the drive specified.

Starting a Process Dump

A process dump is generated and written to a dump file when one of the following conditions occur:

- When your code calls the DosDumpProcess function
- When the DUMPPROCESS=x statement is in CONFIG.SYS and a process (application) encounters an error (trap)
- When FFSTProbe is called with a request for a process dump to be generated

---

FFST Dump

A FFST dump contains information about the processes that were running when the dump was generated along with any user information you requested to be collected.

Setting Up for a FFST Dump

The setup for FFST dumps is done by using the *dumpUserData* and *probe_flags* parameters of the FFSTProbe function.

Refer to Problem Determination APIs for information on using the parameters.

Starting a FFST Dump

When the FFSTProbe function is called and the *dumpUserData* and *probe_flags* parameters have been specified, a FFST dump is generated and stored.

Refer to Problem Determination APIs for information on using the parameters.

Using the PM Dump Facility Dump Formatter

You use the PM Dump Facility dump formatter to display system dump, process dump, or FFST dump information. The system dump contains data about all system activities. The process dump contains data about a single process. The FFST dump contains the captured information your code requested in the call to FFSTProbe.

Starting the PM Dump Facility Dump Formatter

The PM Dump Facility dump formatter can be accessed by using one of the following:

- By clicking on the **PM Dump Facility** in the **Problem Determination Tools** folder. You then click on the **File** action-bar item, select the **Open** menu choice, and select the dump file you want to format.
- From the SYSLOG Details window by selecting the **Tools** menu-bar option (see Displaying Error Log Entry Data).
- By dragging a dump file and dropping it onto the PM Dump Facility icon.

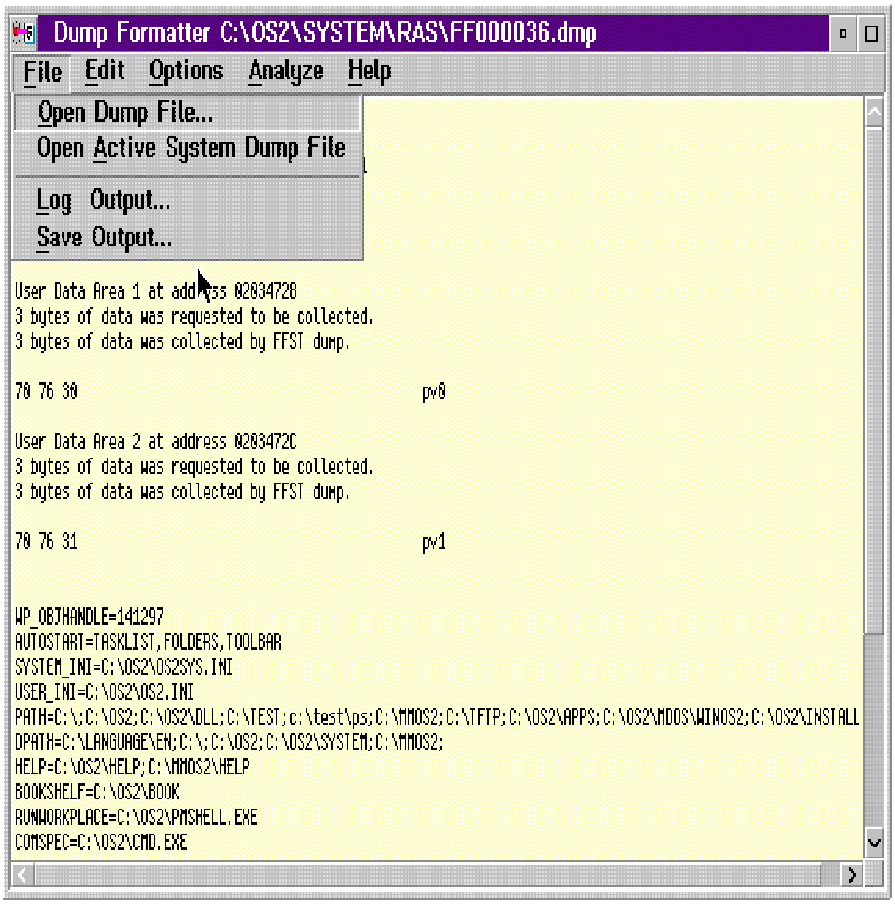
  When you drop multiple copies of the dump files to the PM Dump Facility icon, the dump formatter opens one window for each copy of the dump.
- By selecting **Dumps** from the **Actions** menu-bar option on the FFST setup window.

When you start the PM Dump Facility dump formatter without an initial dump file, a window appears with four action-bar choices: **File, Edit, Options,** and **Help.**. Use the **File** option to select the dump file you want to view. When you open the file, the **Analyze** option appears.

Selecting a Dump to Analyze

You use the **File** option to open a system dump file.

File Menu-Bar Choices on PM Dump Facility Dump Formatter Window



PM Dump Facility Dump Formatter File Option

The **File** pull-down menu offers the following options:

Open Dump File

This option prompts you for the dump file name to be analyzed. You can also open a dump file by dragging it and dropping it onto the client window of the PM Dump Facility dump formatter. When you drop multiple copies of a dump onto the client window, the system opens only the first copy.

When you analyze a system dump or process dump, a command line will be shown at the bottom of the PM Dump Facility dump formatter display. You can use the formatter commands as a fast way to find specific data. Refer to the *PMDF document* in the Troubleshooting folder for a list of system dump and process dump commands.

When the dump file is first opened, only limited data will appear. Use the **Analyze** option to access the remaining dump data.

Log Output

This option prompts you to start or stop logging output to a file. Log data may be appended to an existing log file.

Save Output

This option allows you to save all output displayed in the PMDF scrollable window.

Decompress Dump File

Select this option to decompress a new dump.

Note: For diskette dumps the DUMPDATA.nnn files may be copied for a directory on the hard drive and decompressed from there.

PMDF has the ability to decompress diskette images created by OS2IMAGE without re-creating the original diskettes. To use this facility each of the image file must be named *image.nnn* where *nnn* is a numeric sequence number that corresponds to the disk number.

Connect

Connect allows PMDF to be used as a terminal emulator to drive a Kernel Debugger session.

Disconnect

Disconnect terminated the communications session with the Kernel Debugger.

The **Log Output...** option displays a window that allows the logging of formatted dump data to a file. When you turn on logging, the system records all dump information formatted during this session in the log file you specified. The system also displays the formatted dump information at the same time. The system stores the data from the time that you started logging until you close the PM Dump Facility dump formatter. You also stop logging by using the Log Output... Stop push-button. If you open another dump file, logging continues.

When you select the **Save Output...** option, the system displays a standard OS/2 **Save As...** window. Specify the file name for the saved data. The system will save all formatted dump output that was generated since the file was opened or since the last *clear screen* was done. The system saves this "snap-shot" of dump data only when requested using this option. The system does not save the data simultaneously like it does with the **Log Output...** option.

PM Dump Facility Dump Formatter Edit Option

The **Edit** pull-down menu offers the following options:

Search String

Locates text within the scrollable window.

Undo

Reverse the previous Edit Cut action.

Copy

Copy marked text to the clip board.

Cut

Move marked text to the clip board.

Clear Screen

Clears the scrollable window of all text.

PM Dump Facility Dump Formatter Options Option

The **Options** pull-down menu offers the following options:

Font Settings

This allows font selection for displayed output.

Function Keys

This provides a menu to predefine function keys as strings of dump formatter command strings. Commands may be separated by a semi-colon.

Terminal Settings

Allows the communications parameters to be specified for when the **Connect** option of the **File** pull-down is selected.

Save Settings

This will save the current options in the PMDF.INI file for use next time PMDF is started.

PM Dump Facility Dump Formatter Analyze Option

The **Analyze** pull-down menu options are different for each type of dump. These options change when you select a different type of dump to format.

The system dump options are:

- System
- Process
- Threads
- Synopses

The process dump options are:

- Registers
- Task Control Blocks
- Local Descriptors
- Virtual Machine Control Blocks
- Memory Objects
- Module Table
- Process Synopsis

The FFST dump options are:

- Process Environment Data
- Process Status Data
- Format Trace Buffer
- Format Process Dump
- User Data
- Error Log Data

If the dump you selected does not contain data for a particular option, you will not be able to select the option.

Analyzing a System Dump

After you select a system dump for formatting, the **Analyze** pull-down menu for system dumps offers four selections. Each selection displays its own menu selection. The menu selections are: **System, Process, Threads,** and **Synopsis.**. Note that this option is shown only after you open a system dump file.

It is important to note that the output from the **Analyze** option needs to interpreted with care. Some options are precise since they follow control block chains. The Physical Device Driver Chain and Kernel Heap are examples of control block chains. Other options depend on correct symbols being loaded. Options that display stacks are more speculative in what they display.

The following System Dump **Analyze** selections are available:

System	The System menu display the following options: • Physical Device Driver • Virtual Device Driver • Interrupt Stack • Program List • Window Info • Open files • Heap Info • Memory • Trace
Process	The Process menu display the following options: • Process Info • Thread Chain • Module Table • Local Descriptors • Memory Objects
Threads	The Threads menus dumps stacks related to a given thread. The following menu is displayed: • Ring 3 / Stack Trace • Ring 2 / Stack Trace • Ring 0 / Stack Trace • Call Gate
Synopsis	This offers a miscellaneous collection of options, the most important of which is the Trap Screen display. The following menu is displayed: • System Synopsis • Process Synopsis • Trap Screen Info • Semaphore Analysis • Desktop State

When you analyze a system dump, a command line will be shown at the bottom of the PM Dump Facility dump formatter display. You can use the formatter commands as a fast way to find specific data. Refer to the *PMDF document* in the Troubleshooting folder for a list of system dump commands.

Analyzing a Process Dump

The PM Dump Facility dump formatter is started when a process dump file is opened by using the **Open** option of the **File** pull-down menu. The date and time of the dump are displayed. If the dump was created because of a trap then the trap number is displayed otherwise the trap number is shown as *ffffffff*. The current thread slot and register are shown last.

Process Dump Analyze Option

The **Analyze** pull-down menu for a process dump differs from the standard PMDF Analyze facility. The following choices are provided:

- Registers
- Task Control Blocks
- Local Descriptors
- Virtual Machine Control Blocks
- Memory Objects
- Module Table
- Process Synopsis

When you analyze a process dump, a command line will be shown at the bottom of the PM Dump Facility dump formatter display. You can use the formatter commands as a fast way to find specific data. Refer to the *PMDF document* in the Troubleshooting folder for a list of process dump commands.

For information on taking and controlling process dumps, refer to the DUMPPROCESS command in the *Command Reference* and the the DosDumpProcess function in the *OS/2 Warp Version 4 CP Reference* . Both references can be found in the Toolkit.

Analyzing a FFST Dump

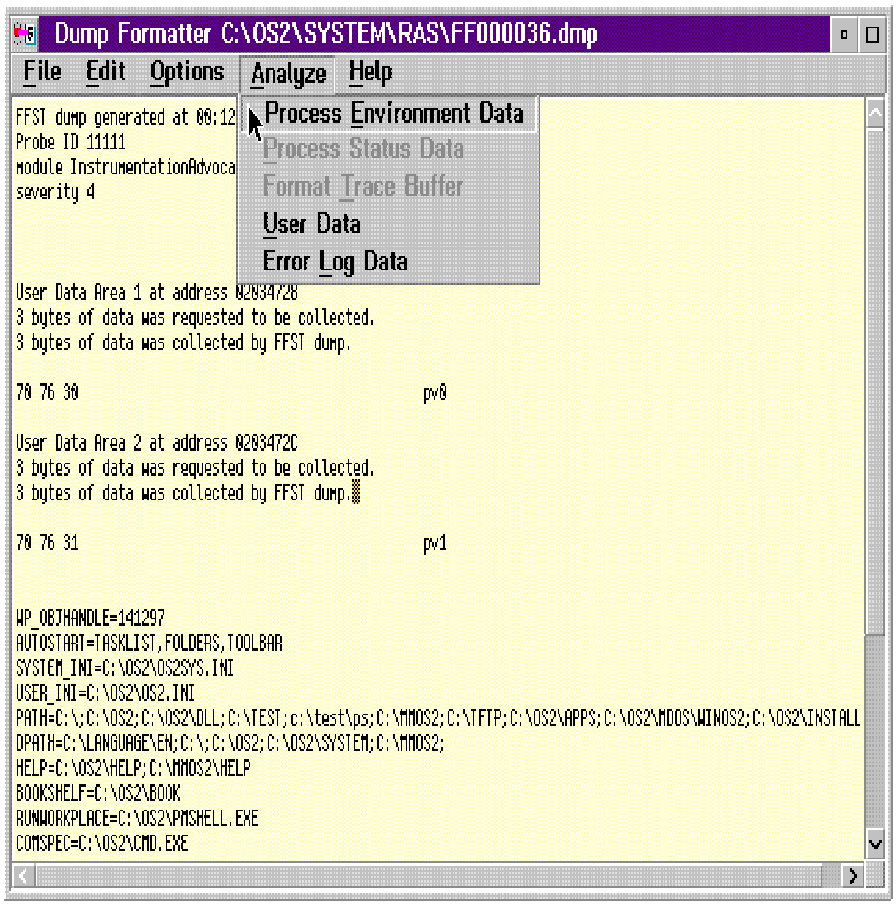
FFST dumps contain information about the processes that were running when the dump was generated along with any user information you requested to be collected.

FFST Dump Analyze Option

The **Analyze** options for formatting FFST dumps are:

- Process environment data
- Process status data
- Format trace buffer
- Format process dump
- User data
- Error log data

Analyze Menu-Bar Choices for a FFST Dump



Process Environment Data

When you select the **Process Environment Data** choice, the PM Dump Facility dump formatter retrieves and displays the environment variables. The system displays the variables that were set at the time the system stored FFST dump. You will not be able to select this option if there is no process environment data in the dump file.

Process Status Data

Select the **Process Status Data** choice to have the PM Dump Facility dump formatter retrieve and display the system process information. The system records all processes that were running on the system when storing the dump. You will not be able to select this option if there is no process status data in the dump file.

Format Trace Buffer

Selecting this option causes the PM Dump Facility dump formatter to start the trace formatter with the full path and name of the trace data file. The PM Dump Facility dump formatter then writes the following message in the window:

```
Format trace buffer in x:\xxxxxxxx.xxx
```

The x:\xxxxxxxx.xxx indicates the path name of the trace file. Then the PM Dump Facility dump formatter starts the trace formatter to format the trace data. The trace data file is a separate file and is not part of the FFST dump file. You will not be able to select this option if the system did not capture trace information when your code called FFSTProbe.

If the system cannot find the path name of the trace data file, use the **File** menu-bar choice of the trace formatter to specify the file name and path name of the trace data file to format.

You can use the PM Dump Facility dump formatter to access the trace formatter. If you close the dump formatter window while the trace formatter window is open, the trace formatter window also closes.

You can find more information on trace and the trace formatter in Analyzing Performance and Debugging Problems Using Trace.

Format Process Dump

Selecting this option causes the PM Dump Facility dump formatter to start another formatter session with the full path and name of the process dump file. This is similar to the way that the trace formatter is accessed.

The PM Dump Facility dump formatter then writes the following message in the window:

```
Formatting process dump file x:\xxxxxxxx.xxx
```

The x:\xxxxxxxx.xxx indicates the path name of the dump file.

If the system cannot find the process dump data file, a *File not found* error message will be displayed. Use **Open** option of the **File** pull-down menu to specify the file name and path name of the dump file to format.

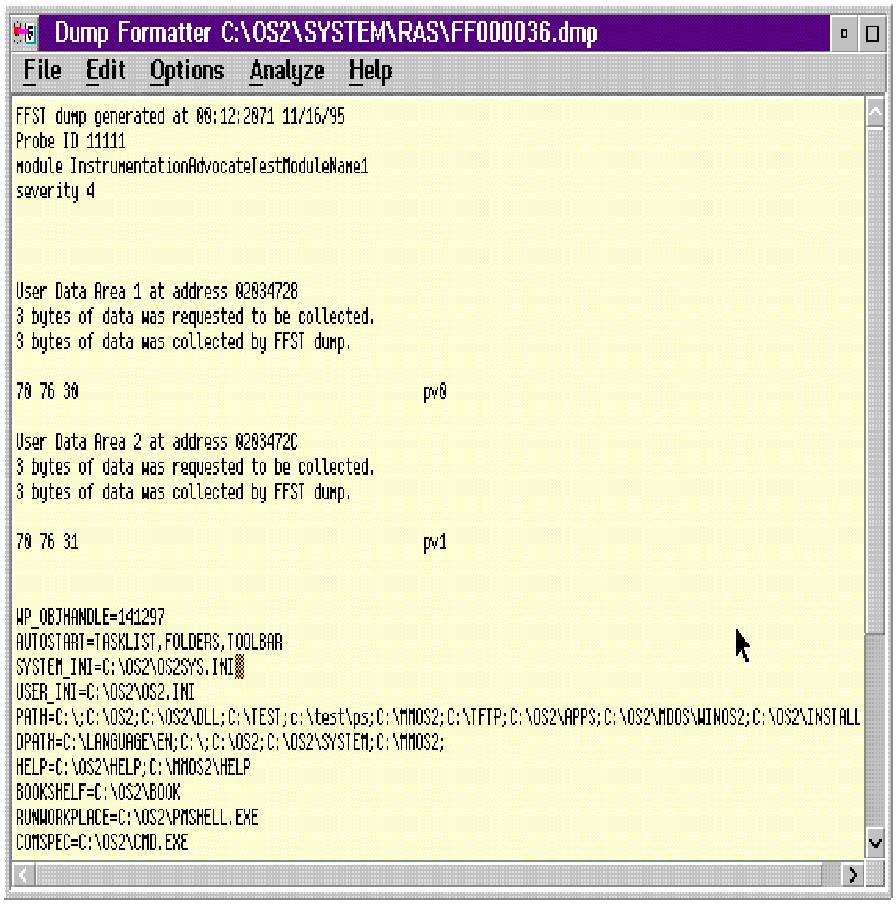
If you close the PM Dump Facility FFST dump formatting window while the process dump window is open, the process dump window also closes.

User Data

When you open the dump and there is at least one area of user data in the dump, the **User Data** option is available from the **Analyze** pull-down menu. You will not be able to select this option if the FFST dump contains no user data.

When you select this option, the PM Dump Facility dump formatter reads the data and displays the data areas one by one. The system displays the data with a label for each data area; for example, *Data Area xxx* (xxx is from 1 to 30).

User Data in PM Dump Facility Dump Formatter Window

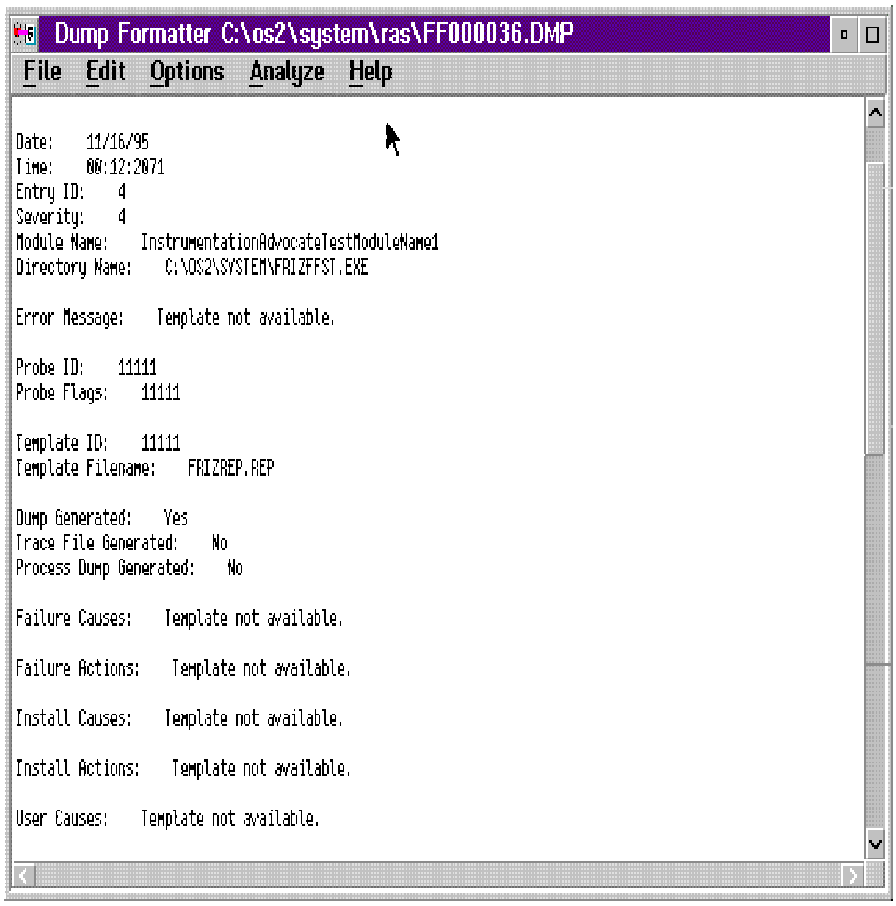


Error Log Data

When you select the **Error Log Data** choice, the PM Dump Facility dump formatter reads the error data from the dump file. The dump formatter then displays the error log entry in the same format as the SYSLOG Details record. There is always Error Log data in an FFST dump. The system always allows you to select this option.

You can find more information on error logs in Viewing and Analyzing Error Log Entries.

Error Log Data in PM Dump Facility Dump Formatter Window



Other PM Dump Facility Dump Formatter Features

The PM Dump Facility dump formatter has a command interface where you can use your own REXX programs to assist in formatting the dump data.

The PM Dump Facility dump formatter also allows you to to use the standard CUA mouse selection and highlighting to select items from the formatted dump.

The PM Dump Facility Mouse Options

Standard CUA mouse selection and highlighting are supported in the PM Dump Facility dump formatter. Use the mouse options to drag and drop marked items onto the command line of the system dump or process dump window.

A double-click with mouse button 1 will highlight a blank delimited string.

A double-click with mouse button 2 will display a pop-up menu of items to use for processing highlighted dump data.

These selections provide the same function as some of the dump formatter commands. They are intended to provide a easy way to perform the same function as the command without having to enter the command on the command line.

Note: There is no validation done between the data that is highlighted and the menu option you select. You must ensure that the correct data is highlighted for the menu option you select.

Listed below are the mouse option selections available system and process dumps:

- List Near
- UnAssemble
- Selector Info
- Unwind Stack
- Thunk Address
- Display Memory
    - Bytes
    - Words
    - Dwords
    - ASCII
- Structures
    - ExEntry_s
    - memstat_s
    - ptda_s
    - tcb_s
    - vmah_s
    - _WND
    - _MQ
    - KSEMSHR
    - KSEMMTX
    - KSEMEVT
    - RamSemStruc
    - SysSemTblStruc
    - Unlisted
- Chains
    - ExEntry_s
    - ˜ptda_s
    - ˜tcb_s
    - ˜vmah_s
    - _˜MQ
    - _˜WND
    - ˜Unlisted

Summary

Set up and triggering for FFST dumps are different than set up and triggering for system dumps.

The system creates FFST dumps when the FFSTProbe function requests user data, process environment data, or process status data. Running a FFSTProbe always causes an entry to be made in the system-error log. A FFST dump does not shut the system down.

A process dump contains limited information about a single process that was running at the time of the failure.

The system creates system dumps when encountering a serious problem or error. When the system creates a system dump, the contents of main memory is stored in the system dump file and the system shuts down.

Use the PM Dump Facility dump formatter to format, display, and analyze FFST dumps, process dumps, and system dumps.

The Desktop Management Interface (DMI)

This chapter provides background about the DMI, information about the relationship between DMI and VPD (vital product data) and coding examples.

DMI Overview

The Desktop Management Interface (DMI) provides the means for software and hardware components to define VPD information. DMI also

provides a standard function set that management applications can use to access that information.

The DMI consists of four elements:

- Format for describing information
- Format for data transfer
- Mechanism for data transfer
- Set of services for facilitating communication

You define component descriptions in a language that is called the Management Information Format (MIF). Each component has a MIF file that describes the manageable characteristics of the component.

Component providers use the Component Interface (CI) to describe how to access management information and enable a component to be managed.

Applications use the Management Interface (MI) to manage components.

The CI and MI are data interfaces, as opposed to procedural interfaces.

The Service Layer (SL) is an active, resident code that runs on the system. The SL mediates between the MI and the CI and performs services on behalf of each.

The DMI is a local interface that is used within a single system. DMI does not replace existing network management protocols. DMI provides a consistent method for providing instrumentation to those protocols. The Service Layer is the broker of local instrumentation.

DMI provides the following:

- Independence of a specific computer, operating system, or management protocol
- An easily adopted interface for application developers
- Does not require a network
- Mapping to existing management protocols.

The DMI does not address or specify a protocol for management over a network.

DMI Structure

The DMI is a local interface for single-system use, regardless of being a standalone desktop system or part of a network. The interface consists of the following parts:

- Service Layer (SL): A local program that collects and manages product information in the MIF database. The Service Layer distributes requested information to management applications through the Management Interface (MI) and to manageable products through the Component Interface (CI).

- MIF database: The database containing the installed or attached manageable product information. MIF files contain the information, and the Service Layer manages the information.

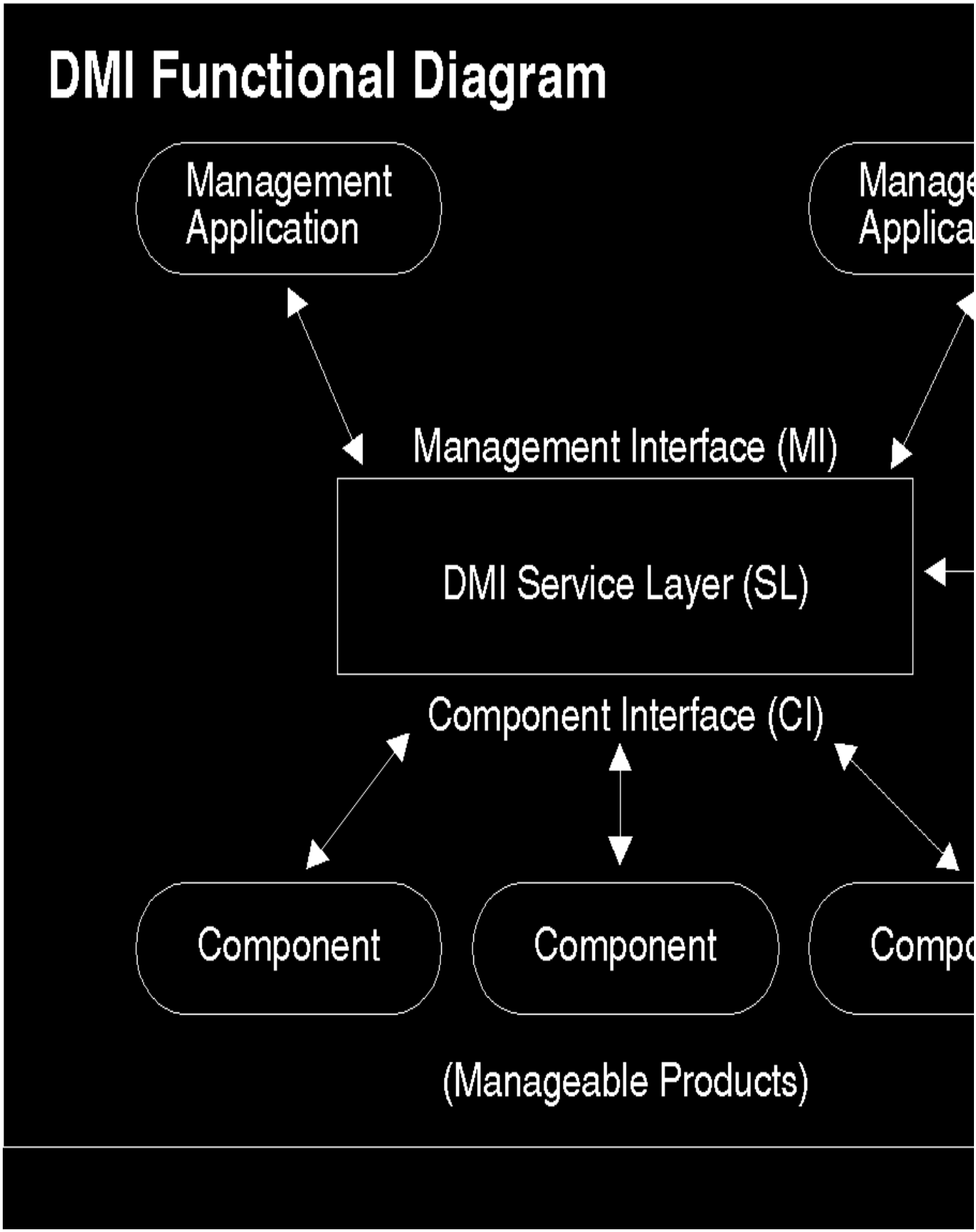
- Management applications: Remote or local programs used to interrogate, track, control, and list the elements of a desktop system. A management application can be any of the following:

    - a graphical user interface program
    - a network management agent
    - an installer program
    - a diagnostics program
    - a remote procedure call.

- Manageable products: Components that are connected to or part of a desktop computer system or network server. Components can be part of the system code, or you can add them later. Each product has a MIF file in the MIF database that contains pertinent product management information.

The following figure shows the DMI structure:

DMI Functional Diagram


DMI Functional Diagram
Management Application
Manage Applica
Management Interface (MI)
DMI Service Layer (SL)
Component Interface (CI)
Component
Component
Compo
(Manageable Products)

Enabling Manageable Products for DMI

The product manufacturer needs to decide which attributes of the product are manageable. To enable a product for DMI you write a MIF file that contains descriptions of these manageable attributes. Group the attributes logically so you can write instrumentation code to provide attribute data.

Use the *SystemView Agent Programmer's Guide* in the Toolkit in the Problem Determination Tools folder as a reference for enabling products. This book defines requirements for products that communicate with the Service Layer. To develop such a product:

1. Read the *SystemView Agent Programmer's Guide* to get an understanding of how components interface with the Service Layer and the supported commands.
2. Define manageable attributes and specific features of your product. Look for existing MIF groups that may apply to your product.
3. Begin with the ComponentID group and define MIF groups that are based on the features of your product. Use as many of the existing standard groups that apply to your product as is possible. Standard groups are available from the Desktop Management Task Force (DMTF). If needed, add proprietary groups for unique features of the product.
4. Decide if the attributes are read-only, read-write, or write-only, and define MIF attributes for each group.
5. Use the syntax that is defined in the *SystemView Agent Programmer's Guide* to create your MIF file.
6. Determine whether to use run-time programs or the direct interface to provide component information to the Service Layer. Write your instrumentation code accordingly.
7. Change the programs that install or delete MIF database file for your component.
8. Decide which errors, exceptions, or problems are to be sent to the Service Layer as indications and the attribute information to include. Real-time, intelligent product management is made possible by using indications because the system immediately notifies management applications when a problem occurs. Add code to your product to send these indications to the Service Layer. The system uses the DmiIndicate command block to pass events to the Service Layer. See the *SystemView Agent Programmer's Guide* in the Problem Determination Tools folder on the system for information about the DmiIndicate command block.

Designing MIF Files

MIF files are the key to enabling your product to be managed by the desktop system. A well designed MIF file is important.

The DMI requires that the MIF file be an ASCII text file that contains the groups and attributes that you defined for your product. You must provide all meaningful information by using the MIF file. The text in the file must follow the defined syntax and grammar.

After you decide which product attributes are manageable, define the attributes into standard groups as much as possible in the MIF file.

The DMI Specification describes attribute characteristics and options. All attributes must have these four items defined:

Name	The name of the attribute
ID	The unique, sequential attribute ID within the group containing the attribute
Type	The attribute's data type. The type is either a specific data type that DMI supports or an enumerated list that provides flexibility in the attribute definitions
Value	The actual value (to be put into the database) or a pointer to the instrumentation that gets the value

In addition, you can optionally define description, access, and storage attributes.

The description can be one or many lines. The description text should be as clear and informative as possible to provide management applications with complete information about the attribute. You need a good description at the component level and group level.

The system uses an access statement to determine whether the attribute's value is read-only, read-write, or write-only.

The storage statement provides a hint to management applications about whether to assist in optimizing storage requirements. Two options are available:

- *common* - you use this option to signify that the attribute value is typically limited to a small set of possibilities that you can be optimize.
- *specific* - you use this option to signify that the attribute value is not a good candidate for optimization because of the possibility of a large number of different values.

You can find the MIF file requirements in the *SystemView Agent Programmer's Guide* in the Toolkit in the Problem Determination Tools folder.

MIF Attribute Storage

The system stores product attribute information in a MIF file. The two ways to store this information are as follows:

- Hard coding the information in the MIF database.
- Keeping the information in your own storage location (for example, a database or in read-only memory (ROM)).

Hard Coding Information into a MIF Database

The simplest way to make product information known through DMI is by hard coding the information into the MIF database. Using this method requires no access code to be written, and DMI retrieves the product information from the DMI database.

Hard-coded information has a major limitation: the system cannot dynamically update the information yet maintain data integrity. If you set an attribute to read/write, anyone can update that value. Another restriction of hard-coded information is that you cannot add or delete rows in tables. The only way to update a read-only attribute is by deleting the old MIF file and installing a new MIF file with the updated values.

Storing Information in Your Own Storage Location

Instrumentation is program code that the Service Layer starts. The code can be a run-time program that requests attribute values. The code could also be a direct interface program. The direct interface program runs on the system and is connected to the Service Layer.

Use run-time programs for products that provide transitory data or that are not associated with a device driver that is loaded in RAM. Use run-time programs for background tasks such as the system BIOS.

Use direct interface programs when the system requires persistent data for products that use device drivers or background programs.

All instrumentation code should include the header file DMIAPI.H.

Instrumentation code that the Service Layer starts must provide one value at a time. The code must be able to handle the following DMI commands that are accessed through the DmiCiInvoke call from the Service Layer:

- DmiGetAttributeCmd
- DmiSetAttributeCmd
- DmiGetRowCmd
- DmiGetFirstRowCmd
- DmiGetNextRowCmd

Every command must call the pConfirmFunc function, including the set commands and the commands that generate errors. When you use the DmiGetRowCmd command, the Service Layer builds the DmiGetRowCnf structure prior to calling DmiCiInvoke. In addition, the GroupKeyData structures are allocated and the oGroupKeyList slot in the DmiGetRowCnf structure is filled in. Any instrumentation code you provide must change only the confirm buffer, not the DmiInvoke command buffer.

The DMI procedure library (DMIAPI) is part of the Toolkit and provides a simple method of handling these constraints. The Component Interface outline makes use of DMIAPI. If you use the outline program as the basis for your instrumentation program, you need only write the

five outlined procedures. When you use the outline program, you do not need to consider any of the issues above.

Dynamic Link Libraries (DLL)

The Component Interface outline lists five procedures that define the simplest method for you to write run-time instrumentation. When you complete the outline files, include them in the library and in your program.

To use run-time programs, use an instrument to specify the attribute value of the MIF Path block name. The block name should contain the run-time program file name or path name. (See the *SystemView Agent Programmer's Guide* on the system for details about using the Path block in the MIF file.) This program runs when the Service Layer needs to access the attribute values.

If the file name in the path block does not have a complete path, the Service Layer uses the library path to find the DLL.

Direct Interface (DI) Programs

Your component must register with the Service Layer before it can use the direct interface. You specify the keyword DIRECT-INTERFACE in the Path block of the MIF file. If the direct interface program is not running, the system returns an error when the Service Layer attempts to access attributes.

Direct interface components register with the Service Layer using the DmiRegisterCiCmd command. When an attribute value needs to be accessed, the Service Layer passes control to the direct interface program. A direct interface program must use the DmiUnregisterCiCmd to unregister with the Service Layer before the system unloads the program.

Indications

Component instrumentation sends unsolicited messages, to the Service Layer to signal some particular situation. The messages are known as *events* . When the messages reach a management application, they are known as *indications* . The indication ID identifies indications. The IDs start at the number 1. Events are often a sign of a catastrophic occurrence or other activity that the system should know about immediately. The event identification is specific to a given component and appears as an unsolicited "get." You specify event structures as groups in the component's MIF file.

In DMI, all commands are specified by using data blocks. Component instrumentation uses the function call DmiIndicate() to send the indication block and any following blocks to the Service Layer for processing. The C language prototype call is as follows:

```
unsigned long  DmiIndicate(PTR command)
```

where `command` is the complete command block. The system returns a 32-bit status value to indicate success or failure.

The Service Layer immediately returns control to the component instrumentation while it processes the indication. The component instrumentation is free to issue additional indications, but it cannot reuse the original indication block buffer until the Service Layer processing is complete. Simultaneous indications from a component instrumentation must use different indication blocks. When the Service Layer has completed processing the indication, it calls the function pResponseFunc() identified in the original indication block. At that point, the component instrumentation can reuse the block.

Installing MIF Files

When you install your product on a system, the installation process is responsible for putting the component MIF file into the existing MIF database. If the Service Layer is not running, the system stores the component MIF file in the MIF subdirectory.

Install programs use the DmiCiInstallCmd function to install the MIF file in the MIF database.

To accommodate DMI, your installation process must provide an ASCII text MIF file. When you create the MIF file, follow the format that is

found in the *SystemView Agent Programmer's Guide* on the system.

If the Service Layer is not running when you install the MIF, you can copy the MIF file to the <boot-drive>OS2\SYSTEM\RAS\MIFS directory. The next time the system starts the DMI Service Layer, the system installs the MIF file in the MIF database. Then system copies the file to the <boot-drive>OS2\SYSTEM\RAS\BACKUP directory, and deletes the file from the MIFS directory.

Removing MIF Files

To produce a complete DMI-enabled component, you include an uninstall program as well as an install program.

Uninstall programs use the DmiCiUninstallCmd function to remove the MIF file from the MIF database.

Accessing DMI Information

Management applications use the DMI Get, Set, and List commands to request information about manageable products on a local system. Each management application that manages system components must register with the Service Layer to issue commands and to receive notifications of indications.

To register with the Service Layer, an application uses the DmiRegisterMgmtCmd command. To unregister, applications use the DmiUnregisterMgmtCmd command.

Component Information

A DMI component is a hardware or software product that is installed in or attached to a computer system. The system stores information about the component in a MIF file. Use the following List commands to view the information in the MIF database:

- DmiListComponentCmd
- DmiListFirstComponentCmd
- DmiListNextComponentCmd
- DmiListComponentDescCmd

Group Information

DMI organizes attributes into groups that are related to components. DMI requires some groups for all products while requiring other groups for only certain products, such as printers. Other groups may be proprietary and relate only to a specific product. You can view the information that is defined in the MIF database by using the following List commands:

- DmiListGroupCmd
- DmiListFirstGroupCmd
- DmiListNextGroupCmd
- DmiListGroupDescCmd

Attribute Information

An attribute can describe a single characteristic, a component, or a product. Attributes are key parts of a component MIF file. Attributes for

groups vary according to the type of product. To view the information that is defined in the MIF database, use the following List commands:

- DmiListAttributeCmd
- DmiListFirstAttributeCmd
- DmiListNextAttributeCmd
- DmiListAttributeDescCmd

Management applications must provide code to deal appropriately with the status codes that the Service Layer returns. The following table lists the error codes that the Service Layer returns. The table contains the return code, the *#define* used in the VPD, and a list of probable causes for the return code. The list of probable causes is in sequence from most likely to occur to least likely to occur.

Table 1. DMI Return Code Table

RETURN CODE	#define DESCRIPTION
	NON-ERROR CONDITION CODES
0x00000000	SLERR_NO_ERROR
	o Good return code; no errors found
0x00000001	SLERR_NO_ERROR_MORE_DATA
	o Good return code; no errors found, but more data is available
	DATABASE ERRORS
0x00000100	DBERR_ATTRIBUTE_NOT_FOUND
	o An invalid attribute ID was specified on the command being executed
	o An invalid component or group ID was specified on the command being executed
	o Component instrumentation has been specified, but it is not available
0x00000101	DBERR_VALUE_EXCEEDS_MAXSIZE
	o A string is too large (> 508 bytes)
0x00000102	DBERR_COMPONENT_NOT_FOUND
	o An invalid component ID was specified on the command being executed
0x00000103	DBERR_ENUM_ERROR
	o Specifying a value for an enumeration that is not defined in the component
0x00000104	DBERR_GROUP_NOT_FOUND
	o An invalid group ID was specified on the command being executed
	o An invalid component ID was specified on the command being executed
0x00000105	DBERR_ILLEGAL_KEYS
	o Invalid keys specified

	o Invalid key count specified o An invalid component or group ID was specified on the command being executed
0x00000106	DBERR_ILLEGAL_TO_SET
	o The attribute access is read only, so the attribute cannot be set
0x00000107	DBERR_OVERLAY_NAME_NOT_FOUND
	o Currently not in use
0x00000108	DBERR_ILLEGAL_TO_GET
	o The attribute access is write-only, so the attribute cannot be read

Table 1. DMI Return Code Table

RETURN CODE	#define DESCRIPTION
0x00000109	DBERR_NO_DESCRIPTION
	o There is no description for the requested component, group, or attribute o If a component description is being requested, the component ID specified on the command could be invalid o If a group description is being requested, the group or component ID specified on the command could be invalid o If an attribute description is being requested, the attribute, group, or component ID specified on the command could be invalid
0x0000010A	DBERR_ROW_NOT_FOUND
	o For the keys specified, the table row cannot be found o Invalid keys specified o Invalid key count specified o An invalid component or group ID was specified on the command being executed
0x0000010B	DBERR_DIRECT_INTERFACE_NOT_REGISTERED
	o A direct interface is indicated but not registered with the service layer
0x0000010C	DBERR_DATABASE_CORRUPT
	o Currently not in use

0x0000010D	DBERR_ATTRIBUTE_NOT_SUPPORTED o Attribute marked as not supported
0x0000010E	DBERR_LIMITS_EXCEEDED o More than the maximum number of elements allowed in the database (for example, the current limit for components is 256)
	SERVICE LAYER ERRORS
0x00000200	SLERR_BUFFER_FULL o Response buffer is full
0x00000201	SLERR_ILL_FORMED_COMMAND o Confirm buffer length too large o Keys for a table (group) not in order o listcomponent command with group keys but not a class string o An offset in the command points to beyond the end of the command buffer o Confirm buffer length < 512
0x00000202	SLERR_ILLEGAL_COMMAND o Not one of the valid DMI commands
0x00000203	SLERR_ILLEGAL_HANDLE o Invalid management handle
0x00000204	SLERR_OUT_OF_MEMORY o A memory allocation failed. Check this possible cause: - A buffer length set too large

Table 1. DMI Return Code Table

RETURN CODE	#define DESCRIPTION
0x00000205	SLERR_NULL_COMPLETION_FUNCTION o There was no callback function specified on the register, and a command has been executed that will require a callback function o There was no callback or indication functions specified on the register
0x00000206	SLERR_NULL_RESPONSE_BUFFER o On a command that requires a response buffer, one has not been specified

0x00000207 SLERR_CMD_HANDLE_IN_USE

- Currently not in use

0x00000208 SLERR_ILLEGAL_DMI_LEVEL

- The iLevelCheck field on the command is not the same as the level of the DMI service layer

0x00000209 SLERR_UNKNOWN_CI_REGISTRY

- The component instrumentation handle (iCiHandle) is not valid

0x0000020A SLERR_COMMAND_CANCELED

- Currently not in use

0x0000020B SLERR_INSUFFICIENT_PRIVILEGES

- Currently not in use

0x0000020C SLERR_NULL_ACCESS_FUNCTION

- Entry point to the component instrumentation is null

0x0000020D SLERR_FILE_ERROR

- Could not delete a component from the MIF database

0x0000020E SLERR_EXEC_FAILURE

- Currently not in use

0x0000020F SLERR_BAD_MIF_FILE

- The MIF database cannot be opened
- There is an error in the MIF file that is being installed

0x00000210 SLERR_INVALID_FILE_TYPE

- A MIF file that is being installed has an invalid file type

0x00000211 SLERR_SL_INACTIVE

- The service layer cannot be contacted and is probably not running

0x00000212 SLERR_UNICODE_NOT_SUPPORTED

- Currently not in use

0x00000213 SLERR_CANT_UNINSTALL_SL_COMPONENT

- Currently not in use

0x00000214 SLERR_NULL_CANCEL_FUNCTION

- Entry point to the cancel function of the component instrumentation is null

Table 1. DMI Return Code Table

RETURN CODE	#define DESCRIPTION
0x00003000	SLERR_NOT_INITIALIZED - Service layer is not initialized and is not running
0x00003001	SLERR_IPC_CREATE_ERROR - Could not communicate or establish communications with the service layer - On a synchronous invoke, could not get exclusive use of the synchronous interface
0x00003002	SLERR_THREAD_CREATE_ERROR - A thread in the service layer cannot be created
0x00003003	SLERR_QUEUE_CREATE_ERROR - Cannot create the queue for the tasker
0x00003004	SLERR_SL_TERMINATED - Currently not in use
0x00003005	SLERR_CMD_EXCEPTION - A trap occurred in the service layer, probably due to an invalid command
0x00003006	SLERR_SYNC_SETUP_ERROR - Cannot create the event semaphore used on the synchronous dmiinvoke call
0x00003007	SLERR_SL_DLL_MISMATCH - The versions of the Service Layer (SL) and the DMIAPI.DLL are incompatible
0x00003008	SLERR_IPC_ERROR - Cannot set up the shared memory with the Service Layer

The *SystemView Agent Programmer's Guide* on the system has details of the DmiRegisterMgmtCmd command. The Guide also describes the Get, Set, and List commands available to management applications as well as the list of status codes.

DMI Browser

The DMI browser is a desktop tool for managing hardware and software products that conform to the DMI standard. You can use the browser to do the following:

- View hardware or software product information about the components, groups, and attributes
- Install product MIF files in the database
- Remove product MIF files from the database
- Dynamically register and unregister products with the DMI service layer
- View events that the service layer issues
- Display version information

You can use the DMI browser as a test tool during product development. If the browser displays error messages if it detects problems during the installation of MIF files or registration of products.

Summary of Functions and Interfaces

OS/2 Warp Version 4 provides functions, commands, graphical utilities, icons, and folders to help you collect and manage problem determination data. Here is a summary to help you understand all that is available to you, and where more information can be found about each.

The tables appear in the same order as the chapters in the book:

- First Failure Support Technology
- Trace
- Dumps
- Error Log
- DMI

Most of the function described in the tables can be accessed from the systems management folder

Table 2. FFST

NAME	TASKS	FOR INFORMATION
FFSTProbe function	o Logs data in Error Log o Can trigger FFST dump	Instrumenting Your Code
FFSTQueryConfiguration function	Queries FFST configuration information	Instrumenting Your Code
FFSTSetConfiguration function	Change FFST configuration values	Instrumenting Your Code
MKTMPF (Make Template File) command	Creates an error record template file	Instrumenting Your Code
MRES (MultiRes Resource Compiler)	Creates message files	Instrumenting Your Code
FFSTCONF command	Controls error data requested by the FFSTProbe function	Controlling FFSTProbe
FFST Setup icon	Accesses FFST Setup display	Controlling FFSTProbe

Table 3. Trace

NAME	TASKS	FOR INFORMATION
TRACE utility	Turns trace points on and off	Using Trace
TRACEFMT utility	Views error event trace data	Using Trace
TRACEGET command	Captures contents of trace buffer to a file	Using Trace
TRACELOG utility	Controls logging of event trace data	Using Trace
TRCUST utility	Trace Customizer to create trace format file	Using Trace

Table 4. Dump

NAME	TASKS	FOR INFORMATION
PM Dump Facility icon	Initiates the PM Dump Facility dump formatter	Capturing Dumps

Table 5. Error Log

NAME	TASKS	FOR INFORMATION
LogOpenEventNotification function	Requests notification of when entries are put in error log	Problem Determination APIs
LogChangeEventFilter function	Changes the event notification filter	Problem Determination APIs
LogReadEntry function	Reads entries from the log file	Problem Determination APIs
LogOpenFile function	Opens a log file for reading	Problem Determination APIs
LogCloseFile function	Closes a log file	Problem Determination APIs
LogWaitEvent function	Request error notification	Problem Determination APIs
LogCloseEventNotification function	Closes event notification	Problem Determination APIs
LogFormatEntry function	Gets character strings that can be displayed when formatted.	Problem Determination APIs
SYSLOG (Error Log Formatter) utility	Displays the system Error Log Allows access to other	Viewing Error Log

Table 6. DMI

NAME	FUNCTION(S)	FOR INFORMATION
DMI	Standard way and API set to define and access VPD	DMI

Problem Determination APIs

These APIs are intended for programmers developing OS/2 applications. Problem Determination services provide basic support for identifying and isolating errors. They include an integrated Error Log to keep a historical record of errors detected by the operating system and, optionally, by applications. The mechanism for inserting entries into the Error Log is the FFSTProbe function. FFST (First Failure Support Technology) enables errors to be logged when first detected.

FFSTProbe also supports the collection of trace and user-specified information and the association of it with an error log entry and possibly an FFST dump. Traces are used to create a historical record of activity in the operating system and, optionally, applications. FFST dumps are produced if the user calls FFSTProbe with certain parameters filled in.

A system dump is used to create a snapshot of the entire contents of main memory at the time of a system crash. In most cases, a system dump should be taken only with the assistance of a Service Representative.

The information in the Error Log, along with any trace or dump information, can be valuable in isolating errors more quickly and with less disruption to users.

Problem Determination services provide application programming interfaces (APIs) for:

- Managing Error Log files
- Reading and formatting Error Log entries
- Registering and waiting for Error Log event notifications
- Changing Error Log event-notification filters
- Logging an error and storing associated trace and dump information in a file

The following libraries must be linked with object files that use the Problem Determination functions:

Library	Functions
ffst.lib	FFST functions
lfapi.lib	Log functions
trace.lib	TraceCreateEntry

This chapter includes the following sections:

Problem Determination Functions

This sections describes the following Problem Determination functions:

- FFSTProbe
- FFSTQueryConfiguration
- FFSTSetConfiguration
- LogChangeEventFilter
- LogCloseEventNotification
- LogCloseFile
- LogFormatEntry
- LogOpenEventNotification
- LogOpenFile
- LogReadEntry
- LogWaitEvent
- TraceCreateEntry

FFSTProbe

FFSTProbe - Syntax

FFSTProbe performs the requested services and returns control to the caller. This function provides a series of functions that includes logging and dump creation.

```
#define INCL_FFST
#include <os2.h>

PPRODUCTINFO    pProductInfo;
PFFSTPARMS      pFFSTParms;
APIRET          rc;            /*  Return code. */

rc = FFSTProbe(pProductInfo, pFFSTParms);
```

FFSTProbe Parameter - pProductInfo

pProductInfo (PPRODUCTINFO) - input
Pointer to the product-information packet.

FFSTProbe Parameter - pFFSTParms

pFFSTParms (PFFSTPARMS) - input
A parameter packet that describes the information provided with the FFSTProbe.

FFSTProbe Return Value - rc

rc (APIRET) - FFSTProbe returns one of the following values:

- From FFSTProbe to application

```
1000000      No error
```

- Invalid Pointers passed in by the API user

```
1000005      Invalid product address
1000010      Invalid FFST address
1000015      Invalid proddata address
1000016      Invalid message address
1000017      Invalid pmsginsdata address
1000020      Invalid DMI address
1000025      Invalid dump address
1000026      Invalid pdumpuserdata address
1000030      Invalid user area address
1000035      Invalid log data address

1000040      Invalid insert messages address
1000045      Invalid vendor tag address
1000050      Invalid tag address
1000055      Invalid modulename address
1000060      Invalid revision address
1000065      Invalid product id address
1000070      Invalid DMI fixlvl address
1000075      Invalid tempfn address

1000080      Invalid user structure title address
1000085      Invalid config subproddata address
1000090      Invalid DMI modification level address
```

- Invalid Packet Revision Numbers

```
1000095      Invalid product revision
1000100      Invalid DMI revision
1000105      Invalid FFST revision
```

- Invalid severity

```
1000110      Invalid severity
```

- Invalid number of dumps specified by user

```
1000115      Invalid user dump number
```

- Invalid number of insert texts

```
1000120      Invalid inserts number
1000125      Invalid insert text address
```

- Invalid probe flags

```
1000130      Invalid probe flags
```

- Invalid PSTAT data and process environment data

```
1000135      Invalid PSTAT data
```

```
1000140      Invalid process environment
```

- Invalid packet sizes

```
1000145      Invalid PRODUCTDATA packet size
1000150      Invalid DMIDATA packet size
1000155      Invalid FFSTPARMS packet size
```

- DLL load and query

```
1000160      DLL query proc error
1000165      DLL load error
```

- FFSTProbe pipe errors

```
1000170      Client pipe not created
1000175      Client pipe not opened
```

- Multiple errors occurring during probe processing

```
1000180      Multiple system errors
```

- Misc. Return codes

```
1000185      Client proc in exit list processing
1000186      UniCode conversion error
```

- FFSTProbe not active

  Returned to application and not to ERRLOG.

```
1000190      FFST not active
```

  /*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659a       Dump wrap error
0x1659b       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165a1       Create config semaphore error
```

```
0x165a2       Create dump semaphore error
0x165a3       Worker alloc shared mem error
0x165a4       Worker get shared mem error
0x165a5       Create pct semaphore error
0x165a6       File already exists
0x165a7       Specified file not found
0x165a8       Dump validation error
0x165a9       Config memory filled

0x165aa       Worker setconfig error
0x165ab       Worker pipe not created
0x17005       Worker ffst config not okay
0x165ac       Worker mutexsem not released
0x17010       Worker already active
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTProbe - Parameters

pProductInfo (PPRODUCTINFO) - input
Pointer to the product-information packet.

pFFSTParms (PFFSTPARMS) - input
A parameter packet that describes the information provided with the FFSTProbe.

rc (APIRET) - returns
Return code.

FFSTProbe returns one of the following values:

- From FFSTProbe to application

```
1000000       No error
```

- Invalid Pointers passed in by the API user

```
1000005       Invalid product address
1000010       Invalid FFST address
1000015       Invalid proddata address
1000016       Invalid message address
1000017       Invalid pmsginsdata address
1000020       Invalid DMI address
1000025       Invalid dump address
1000026       Invalid pdumpuserdata address
1000030       Invalid user area address
1000035       Invalid log data address

1000040       Invalid insert messages address
1000045       Invalid vendor tag address
1000050       Invalid tag address
1000055       Invalid modulename address
1000060       Invalid revision address
1000065       Invalid product id address
1000070       Invalid DMI fixlvl address
1000075       Invalid tempfn address

1000080       Invalid user structure title address
1000085       Invalid config subproddata address
1000090       Invalid DMI modification level address
```

- Invalid Packet Revision Numbers

```
1000095       Invalid product revision
1000100       Invalid DMI revision
```

```
1000105      Invalid FFST revision
```

- Invalid severity

```
1000110      Invalid severity
```

- Invalid number of dumps specified by user

```
1000115      Invalid user dump number
```

- Invalid number of insert texts

```
1000120      Invalid inserts number
1000125      Invalid insert text address
```

- Invalid probe flags

```
1000130      Invalid probe flags
```

- Invalid PSTAT data and process environment data

```
1000135      Invalid PSTAT data
1000140      Invalid process environment
```

- Invalid packet sizes

```
1000145      Invalid PRODUCTDATA packet size
1000150      Invalid DMIDATA packet size
1000155      Invalid FFSTPARMS packet size
```

- DLL load and query

```
1000160      DLL query proc error
1000165      DLL load error
```

- FFSTProbe pipe errors

```
1000170      Client pipe not created
1000175      Client pipe not opened
```

- Multiple errors occurring during probe processing

```
1000180      Multiple system errors
```

- Misc. Return codes

```
1000185      Client proc in exit list processing
1000186      UniCode conversion error
```

- FFSTProbe not active

  Returned to application and not to ERRLOG.

```
1000190      FFST not active
```

  /*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659a       Dump wrap error
0x1659b       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165a1       Create config semaphore error
0x165a2       Create dump semaphore error
0x165a3       Worker alloc shared mem error
0x165a4       Worker get shared mem error
0x165a5       Create pct semaphore error
0x165a6       File already exists
0x165a7       Specified file not found
0x165a8       Dump validation error
0x165a9       Config memory filled

0x165aa       Worker setconfig error
0x165ab       Worker pipe not created
0x17005       Worker ffst config not okay
0x165ac       Worker mutexsem not released
0x17010       Worker already active
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTProbe - Remarks

The library FFST.LIB must be linked with object files that use FFSTProbe.

The *packet_revision_number* parameter defines if pointers point to ASCII or UniCode character data.

FFSTProbe - Example Code

The following example adds an error (235) to the Default Log file for this service. The error has no message inserts, user data, dump file, or extra data. Product information will be retrieved from the DMI database entry that was created when the product (TEST PRODUCT) was installed.

```
/*********************************************************************/
/* probe.c: FFSTProbe sample                                         */
/*                                                                   */
/* This test program gives an example of using the FFSTProbe API and the  */
/* TraceCreateEntry API by using 'wrapper' functions.  The dummy API      */
/* My_Dummy_Api returns a return code which is then used as the basis of  */
```

```
/* firing a FFSTProbe via the wrapper function, callFFST.  callFFST can   */
/* be modified to include more or less data as needed.                     */
/*                                                                         */
/***************************************************************************/

#define INCL_DOS
#define INCL_DOSMEMMGR
#define INCL_DOSPROCESS
#define INCL_FFST
#define NO_ERROR 0

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <FFST.h>
#include <trace.h>

/***************************************************************************/
/* Define probe ID for FFSTProbe called when dummy API fails.  Probe ID    */
/* is the unique identifier you use later to find the source of the        */
/* failure.  It should be unique within a DMI triplet (explained later)    */
/* or within your product                                                  */
/***************************************************************************/
#define DUMMY_API_PROBE 22222

void callFFST ( ULONG input_version,           /* FFST 'Wrapper' Function */
            /* input version lets you change the wrapper     */
            /* without changing each call, just make sure    */
            /* the wrapper still treats the 'old' version    */
            /* the same and that any new code is             */
            /* conditioned on a new input_version #          */

                ULONG input_probe_flags,         /* FFSTProbe probe flags */
                ULONG input_severity,               /* FFSTProbe severity */
                ULONG input_probe_id,                     /* FFSTProbe ID */
                CHAR* input_module_name,   /* module name passed to probe */
                ULONG input_log_data_length, /* log data length for the
                                                system error log          */
                PVOID input_pError_log_data,  /* pointer to the data for
                                                 system error log         */
                int   argc);

/*****************************************************/
/* This is for a common trace entry routine          */
/*****************************************************/
#define HKWD_TEST            220        /* major code */
#define hkwd_test_entry      0x0001     /* minor code for entry */
#define hkwd_test_exit       0x8001     /* minor code for exit */

struct
{
  int count;
  char textì12 ;
} trace_capture_start, trace_capture_end;

APIRET trace_out(ULONG major, ULONG minor, void *trace_data,
                 ULONG data_len);      /* trace wrapper function */

/*************************************/
/* End of trace declarations for Main */
/*************************************/

ULONG My_Dummy_API(ULONG Mydata);


/***************************************************************************/
/*                                                                         */
/*  Main Application (this uses the callFFST wrapper function).            */
/*                                                                         */
/***************************************************************************/

int main ( int argc, char * argvì , char * envp  )
{
   ULONG  rc          = 0;
   ULONG  Mydata      = 2;
   ULONG  userDataLen = 0;
   PVOID  pUserData   = NULL;
```

```
   printf ( "Starting FFSTProbe Sample \n" );

/************************************/
/* Do the trace entry point         */
/************************************/

   trace_capture_start.count = 3; /* just a number */
   strncpy(trace_capture_start.text, "Start main", 12);

/******** CALL TraceCreateEntry function ****/
   trace_out(HKWD_TEST,
      hkwd_test_entry,
      &trace_capture_start,
      sizeof(trace_capture_start));

   /*********************************************************************/
   /* call the 'dummy'  API so it returns a non-zero rc                 */
   /*********************************************************************/
   rc = My_Dummy_API ( Mydata );
   if ( rc != NO_ERROR )
   {
      /*********************************************************************/
      /* The API has failed. Setup the userData to contain the failing rc */
      /*********************************************************************/
      pUserData = calloc ( 2, sizeof ( ULONG ) );
      memcpy ( pUserData, &rc, sizeof ( ULONG ) );
      memcpy ( ( PBYTE ) pUserData + sizeof ( ULONG )
             , &Mydata, sizeof ( ULONG ) );
/*************************************************************************/
/* Call the FFSTProbe wrapper function with a version of 1,              */
/* Have FFST post the process status and environment variables in the    */
/* syslog, a severity of 4, a probe id of DUMMY_API_PROBE which was      */
/* previously defined as 22222, a module name of 'my_module_1', the length */
/* of logusrdta, the logUserData (equal to the failing rc (1) as         */
/* setup above) and Argc is passed in to determine whether or not data   */
/* should be retrieved from DMI.                                         */
/*************************************************************************/


         callFFST ( 1
               , PSTAT_FLAG   PROC_ENV_FLAG
               , SEVERITY4
               , DUMMY_API_PROBE
               , "my_module_1"
               , 2 * sizeof ( ULONG )
               , pUserData
               , argc );
   }

   if (pUserData != NULL)
   {
     free(pUserData);
     pUserData = NULL;
   }

   if (argc > 1)
   {
      printf("\nFFSTProbe sample ended not using DMI component:\n\n\n");
   }
   else
   {
      printf("\nFFSTProbe sample ended using DMI component:\n\n\n");
   }

/************************************/
/* Do the trace end point           */
/************************************/

   trace_capture_end.count = 99;
   strncpy(trace_capture_end.text, "End main", 12);

/******** CALL TraceCreateEntry function ****/
   trace_out(HKWD_TEST,
      hkwd_test_entry,
      &trace_capture_end,
      sizeof(trace_capture_end));

   return 0;
}
```

```
/*********************************************************************/
/* callFFST is the FFSTProbe wrapper function. It allows you to code the  */
/* FFSTProbe API once with data that is static as far as your usage is    */
/* concerned and allows you to pass in dynamic data.   It also helps      */
/* insulate your code if you decide to change your 'static' options       */
/*********************************************************************/

void callFFST ( ULONG input_version,           /* FFST 'Wrapper' Function */
                ULONG input_probe_flags,         /* FFSTProbe probe flags */
                ULONG input_severity,               /* FFSTProbe severity */
                ULONG input_probe_id,                     /* FFSTProbe ID */
                CHAR* input_module_name,   /* module name passed to probe */
                ULONG input_log_data_length, /* log data length for the
                                                 system error log          */
                PVOID input_pError_log_data,   /* pointer to the data for
                                                    system error log       */
                int   argc)
{
  APIRET  rc = 0;
  PVOID   pvar_n0;
  ULONG   pvar_n1;

  /*********************************************************************/
  /* FFSTProbe API structures.    Described in the API Guide             */
  /*********************************************************************/
  FFSTPARMS     FFSTParms;
  PRODUCTINFO   productInfo;
  PRODUCTDATA   productData;
  DMIDATA       DMIData;
  DUMPUSERDATA  dumpUserData;
  MSGINSDATA    msgInsData;

  /*********************************************************************/
  /* The PRODUCTDATA structure defines the DMI triplet which allows      */
  /* additional product information, including template repository       */
  /* filename, to be retrieved from DMI.   DMI is a industry standard    */
  /* for desktop mgt                                                     */
  /*********************************************************************/
  productData.packet_size            = sizeof ( productData );
  productData.packet_revision_number = PRODUCTDATA_ASCII;
                                       /* data can be ASCII or UNI */
  productData.DMI_tag                = "FFSTProbe Sample";
                                     /* Customize for your program */
  productData.DMI_vendor_tag         = "IBM";
                                      /*Customize for your company */
  productData.DMI_revision           = "1.00";             /* Customize  */

  /*********************************************************************/
  /* The DMIDATA structure below is the information which can either be  */
  /* retrieved by DMI or passed in by the FFSTProbe function. The        */
  /* preferred method is to use DMI. In the example below, you can see   */
  /* the use of either depending on whether or not a parm was passed on  */
  /* call to this program                                                */
  /*********************************************************************/

  if ( !(argc > 1) )
  {
     /*******************************************************************/
     /* Setting this structure to NULL indicates that the information is */
     /* to be retrieved from DMI using the DMI triplet as defined in the */
     /* productData structure.  This is the preferred method.           */
     /* Other files in this example show how to build your own DMI      */
     /*******************************************************************/
     productInfo.pDMIData = NULL;

     /*******************************************************************/
     /* Note: This shows the usage of message insert text and is NOT part*/
     /* of the information that could or could not be retrieved from DMI */
     /* This is included as an example of MsgInsTxt and how it can be    */
     /* used to send probe specific data to the SYSLOG (System Error Log)*/
     /*******************************************************************/
     msgInsData.MsgInsTxti0 .insert_number = 1;
     msgInsData.MsgInsTxti0 .insert_text   = "We did use a DMI component";
  }
  else
   {
```

```
  /*********************************************************************/
  /* fill the DMI data structure - useful only in test environments   */
  /*********************************************************************/
  DMIData.packet_size              = sizeof ( DMIData );
  DMIData.packet_revision_number   = DMIDATA_ASCII;
                                        /* could be unicode instead */
  DMIData.DMI_product_ID           = "FFST_toolkt_sample";
                                   /* note this is different than tag */
  DMIData.DMI_modification_level   = "000000";
  DMIData.DMI_fix_level            = "010101";
  DMIData.template_filename        = "PROBE.REP";
                                    /* this file must be on the DPATH */
  DMIData.template_filename_length = strlen (DMIData.template_filename)
                                         * sizeof ( char );
                                   /* since ascii is being used */
  productInfo.pDMIData             = &DMIData;

  /*********************************************************************/
  /* Note: This shows the usage of message insert text and is NOT a   */
  /* of the information that could or could not be retrieved from DMI */
  /*********************************************************************/
  msgInsData.MsgInsTxtì0 .insert_number = 1;
  msgInsData.MsgInsTxti0 .insert_text   = "We did not use a DMI component";
 }

/*********************************************************************/
/* set the pointers up for PRODUCTINFO                               */
/*********************************************************************/

productInfo.pProductData = &productData;   /* This points to the DMI
                                              related data            */

/*********************************************************************/
/* set up some DUMPUSERDATA items                                    */
/*********************************************************************/
pvar_n0 = "Dump user data";              /* Anything can be dumped
                                            here up to 32 Kbytes       */
pvar_n1 = 2;
dumpUserData.no_of_variables = 2;
dumpUserData.DumpDataVarì0 .var_n_length = strlen(pvar_n0) + 1;
dumpUserData.DumpDataVarì0 .var_n        = pvar_n0;
dumpUserData.DumpDataVarì1 .var_n_length = sizeof(ULONG);
dumpUserData.DumpDataVarì1 .var_n        = (PVOID)(&pvar_n1);

/*********************************************************************/
/* set up a couple of MSGINSDATA messages- just to show it can be done */
/*********************************************************************/
msgInsData.no_inserts   = 2;
msgInsData.MsgInsTxtì1 .insert_number = 2;
msgInsData.MsgInsTxtì1 .insert_text   = "Message insert variable 2";

/*********************************************************************/
/* set the FFSTPARMS structure, most values from DEFINEs above.      */
/* See API GUIDE for details on each field and their possible values   */
/*********************************************************************/
FFSTParms.packet_size            = sizeof ( FFSTParms );
FFSTParms.packet_revision_number = FFSTPARMS_OS2_ASCII;
                                          /* ASCII vs UNICODE data */
FFSTParms.module_name            = input_module_name;
FFSTParms.probe_ID               = input_probe_id;
FFSTParms.severity               = input_severity;
FFSTParms.template_record_ID     = input_probe_id;
FFSTParms.pMsgInsData            = &msgInsData;
FFSTParms.probe_flags            = input_probe_flags;
FFSTParms.pDumpUserData          = &dumpUserData;
                                  /* dump data is stored in .DMP files */
FFSTParms.log_user_data_length   = input_log_data_length;
FFSTParms.log_user_data          = input_pError_log_data;
                    /* log data is stored as part of the SYSLOG entry */

/*********************************************************************/
/* Call the FFSProbe API                                             */
/*********************************************************************/
if ( input_version == 1)
{
   rc = FFSTProbe ( &productInfo, &FFSTParms);
}

printf("\n----- Fired the FFSTProbe, rc=%d\n",rc);
```

```
                /* for example only, do not do this in customer level code */

}

/*****************************************************************************/
/*  This is the dummy  API for use in the example.  It can easily set     */
/*  non-zero rc's                                                            */
/*****************************************************************************/

ULONG My_Dummy_API ( ULONG Mydata )
{
   if ( Mydata != 123456 )
   {
      return 1;
   }
   else
   {
      return 0;
   }
}

/*****************************************************************************/
/*  Trace events function                                                    */
/*****************************************************************************/
APIRET trace_out(ULONG major, ULONG minor, void *trace_data, ULONG data_len)
{
  TCEREQUEST packet;
  APIRET      rc;

  packet.packet_size = sizeof packet;      /* Size of packet in bytes       */
  packet.packet_revision_number = TRACE_RELEASE; /* Revision level of trace */
  packet.major_event_code = major;         /* Major code event to be logged */
  packet.minor_event_code = minor;         /* Minor code event to be logged */
  packet.event_data_length = data_len;     /* Length of callers event buffer*/
  packet.event_data = trace_data;          /* Pointer to callers buffer     */

  /* call the TraceCreateEntry function */
  rc = TraceCreateEntry(&packet);
  return rc;
}
```

FFSTProbe - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code

FFSTQueryConfiguration

FFSTQueryConfiguration - Syntax

FFSTQueryConfiguration queries the FFST configuration parameters.

```
#define INCL_FFST
#include <os2.h>

PULONG          buffer_length;
PCONFIGPARMS    pConfigParms;
APIRET          rc;             /*  Return code. */

rc = FFSTQueryConfiguration(buffer_length,
       pConfigParms);
```

FFSTQueryConfiguration Parameter - buffer_length

buffer_length (PULONG) - in/out
Length (in bytes) of the buffer.

- On input, *buffer_length* contains the length of the provided buffer.
- On output, *buffer_length* contains the total number of bytes that were placed in the buffer. If your buffer is too small, then *buffer_length* will be set to the required size, and no data will be written to the buffer.

FFSTQueryConfiguration Parameter - pConfigParms

pConfigParms (PCONFIGPARMS) - input
Pointer to the buffer where the FFST configuration parameters will be written.

FFSTQueryConfiguration Return Value - rc

rc (APIRET) - returns
Return code.

FFSTQueryConfiguration returns the following values:

- From FFSTQueryConfiguration to application

```
1000195      Invalid config user buffer size address
1000200      Invalid config address
1000205      Invalid config revision number
1000210      Invalid config packet size
1000215      Invalid config dump file wrap
1000220      Invalid config dump file directory size
1000225      Invalid config keep dup dump
1000230      Invalid config dump file directory name address
1000235      Invalid config dump file directory name length
1000240      Invalid config no disabled products
1000245      Invalid config disabled products address
1000250      Invalid config vendor tag address
1000255      Invalid config tag address
1000260      Invalid config revision address
```

```
1000265      Invalid config message popup
```

- Return codes for file errors during Configuration processing

```
0x16570      Config file open error
0x16571      Config file search error
0x16572      Config file write error
```

- Return code for an insufficient buffer passed to Configuration

```
0x1657A      Config insufficient buffer
```

- Return code for invalid dump file directory driver passed to FFSTSetConfiguration.

```
0x1657B      Invalid config dump file dir drive
```

- Return codes for an invalid drive or path passed to FFSTSetConfiguration.

```
0x1657C      Invalid drive requested
0x1657D      Invalid path requested
```

/*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659A       Dump wrap error
0x1659B       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165A1       Create config semaphore error
0x165A2       Create dump semaphore error
0x165A3       Worker alloc shared mem error
0x165A4       Worker get shared mem error
0x165A5       Create pct semaphore error
0x165A6       File already exists
0x165A7       Specified file not found
0x165A8       Dump validation error
0x165A9       Config memory filled

0x165AA       Worker setconfig error
0x165AB       Worker pipe not created
0x17005       Worker FFST config not okay
0x165AC       Worker mutexsem not released
0x17010       Worker already active
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTQueryConfiguration - Parameters

buffer_length (PULONG) - in/out
Length (in bytes) of the buffer.

- On input, *buffer_length* contains the length of the provided buffer.
- On output, *buffer_length* contains the total number of bytes that were placed in the buffer. If your buffer is too small, then *buffer_length* will be set to the required size, and no data will be written to the buffer.

pConfigParms (PCONFIGPARMS) - input
Pointer to the buffer where the FFST configuration parameters will be written.

rc (APIRET) - returns
Return code.

FFSTQueryConfiguration returns the following values:

- From FFSTQueryConfiguration to application

```
1000195      Invalid config user buffer size address
1000200      Invalid config address
1000205      Invalid config revision number
1000210      Invalid config packet size
1000215      Invalid config dump file wrap
1000220      Invalid config dump file directory size
1000225      Invalid config keep dup dump
1000230      Invalid config dump file directory name address
1000235      Invalid config dump file directory name length
1000240      Invalid config no disabled products
1000245      Invalid config disabled products address
1000250      Invalid config vendor tag address
1000255      Invalid config tag address
1000260      Invalid config revision address
1000265      Invalid config message popup
```

- Return codes for file errors during Configuration processing

```
0x16570      Config file open error
0x16571      Config file search error
0x16572      Config file write error
```

- Return code for an insufficient buffer passed to Configuration

```
0x1657A      Config insufficient buffer
```

- Return code for invalid dump file directory driver passed to FFSTSetConfiguration.

```
0x1657B      Invalid config dump file dir drive
```

- Return codes for an invalid drive or path passed to FFSTSetConfiguration.

```
0x1657C      Invalid drive requested
0x1657D      Invalid path requested
```

/*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
```

```
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659A       Dump wrap error
0x1659B       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165A1       Create config semaphore error
0x165A2       Create dump semaphore error
0x165A3       Worker alloc shared mem error
0x165A4       Worker get shared mem error
0x165A5       Create pct semaphore error
0x165A6       File already exists
0x165A7       Specified file not found
0x165A8       Dump validation error
0x165A9       Config memory filled

0x165AA       Worker setconfig error
0x165AB       Worker pipe not created
0x17005       Worker FFST config not okay
0x165AC       Worker mutexsem not released
0x17010       Worker already active
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTQueryConfiguration - Remarks

The library FFST.LIB must be linked with object files that use FFSTQueryConfiguration

The *packet_revision_number* parameter defines if pointers point to ASCII or UniCode character data.

FFSTQueryConfiguration - Related Functions

- FFSTSetConfiguration

FFSTQueryConfiguration - Example Code

The following example will query for FFST configuration. FFST configuration values are returned in the buffer. You can access individual parameters by using the CONFIGPARMS structure.

```
  #define INCL_FFST
  #include <unichar.h>
  #include <os2.h>
  APIRET rc;
  ULONG buffer_length;
  UniChar config_bufferø300!;

  CONFIGPARMS  FFSTConfig;
  PCONFIGPARMS  pFFSTConfig  *FFSTConfig;

  buffer_length = 300;
  rc = FFSTQueryConfiguration (  &buffer_length, &config_buffer );
  if (rc |= 0)                                    /* If Problem      */
          {                                       /* reason          */
          printf("FFSTConfigure error: return code = %d",rc);
          return;
          }
```

FFSTQueryConfiguration - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions

FFSTSetConfiguration

FFSTSetConfiguration - Syntax

FFSTSetConfiguration sets the FFST configuration parameters.

You should use FFSTQueryConfiguration to obtain the current settings before using this function. This allows you to change the desired parameters without affecting others.

```
#define INCL_FFST
#include <os2.h>

PCONFIGPARMS    pConfigParms;
APIRET          rc;            /*  Return code. */

rc = FFSTSetConfiguration(pConfigParms);
```

FFSTSetConfiguration Parameter - pConfigParms

pConfigParms (PCONFIGPARMS) - input
Pointer to FFST configuration parameters.

FFSTSetConfiguration Return Value - rc

rc (APIRET) - returns
Return code.

FFSTSetConfiguration returns the following values:

- From FFSTSetConfiguration to application

```
1000195      Invalid config user buffer size address
1000200      Invalid config address
1000205      Invalid config revision number
1000210      Invalid config packet size
1000215      Invalid config dump file wrap
1000220      Invalid config dump file directory size
1000225      Invalid config keep dup dump
1000230      Invalid config dump file directory name address
1000235      Invalid config dump file directory name length
1000240      Invalid config no disabled products
1000245      Invalid config disabled products address
1000250      Invalid config vendor tag address
1000255      Invalid config tag address
1000260      Invalid config revision address
1000265      Invalid config message popup
```

- Return codes for file errors during Configuration processing

```
0x16570      Config file open error
0x16571      Config file search error
0x16572      Config file write error
```

- Return code for an insufficient buffer passed to Configuration

```
0x1657A      Config insufficient buffer
```

- Return code for invalid dump file directory driver passed to FFSTSetConfiguration.

```
0x1657B      Invalid config dump file dir drive
```

- Return codes for an invalid drive or path passed to FFSTSetConfiguration.

```
0x1657C      Invalid drive requested
0x1657D      Invalid path requested
```

/*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659a       Dump wrap error
0x1659b       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165a1       Create config semaphore error
0x165a2       Create dump semaphore error
0x165a3       Worker alloc shared mem error
0x165a4       Worker get shared mem error
0x165a5       Create pct semaphore error
0x165a6       File already exists
0x165a7       Specified file not found
0x165a8       Dump validation error
0x165a9       Config memory filled

0x165aa       Worker setconfig error
0x165ab       Worker pipe not created
0x17005       Worker ffst config not okay
0x165ac       Worker mutexsem not released
0x17010       Worker already active
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTSetConfiguration - Parameters

pConfigParms (PCONFIGPARMS) - input
Pointer to FFST configuration parameters.

rc (APIRET) - returns
Return code.

FFSTSetConfiguration returns the following values:

- From FFSTSetConfiguration to application

```
1000195      Invalid config user buffer size address
1000200      Invalid config address
1000205      Invalid config revision number
1000210      Invalid config packet size
1000215      Invalid config dump file wrap
1000220      Invalid config dump file directory size
1000225      Invalid config keep dup dump
1000230      Invalid config dump file directory name address
1000235      Invalid config dump file directory name length
1000240      Invalid config no disabled products
1000245      Invalid config disabled products address
1000250      Invalid config vendor tag address
```

```
1000255      Invalid config tag address
1000260      Invalid config revision address
1000265      Invalid config message popup
```

- Return codes for file errors during Configuration processing

```
0x16570      Config file open error
0x16571      Config file search error
0x16572      Config file write error
```

- Return code for an insufficient buffer passed to Configuration

```
0x1657A      Config insufficient buffer
```

- Return code for invalid dump file directory driver passed to FFSTSetConfiguration.

```
0x1657B      Invalid config dump file dir drive
```

- Return codes for an invalid drive or path passed to FFSTSetConfiguration.

```
0x1657C      Invalid drive requested
0x1657D      Invalid path requested
```

/*----------*/ /* Return codes from FFSTProbe, FFSTQueryConfiguration,*/ /* FFSTSetConfiguration or other FFST processing. */ /*----------*/

- Return codes for shared memory errors

```
0x16580       Get shared mem error
0x16581       Alloc shared mem error
0x16582       Free shared mem error
```

- Return codes for semaphore errors

```
0x1658A       Semaphore timeout error
0x1658B       Semaphore open error
0x1658C       Semaphore release error
0x1658D       Semaphore close error
0x1658E       Semaphore request error
0x17111       Semaphore g error
```

- Return codes for dump engine processing which may be posted to the SysLog.

```
0x16595       Dump hdr file open error
0x16596       Index file open error
0x16597       Dump file open error
0x16598       Memory allocation error
0x16599       Queryfs error
0x1659a       Dump wrap error
0x1659b       Trace rename error
0x17222       Proc dump rename error
```

- Return codes that may be in the SysLog during Worker bringup

```
0x165a1       Create config semaphore error
0x165a2       Create dump semaphore error
0x165a3       Worker alloc shared mem error
0x165a4       Worker get shared mem error
0x165a5       Create pct semaphore error
0x165a6       File already exists
0x165a7       Specified file not found
0x165a8       Dump validation error
0x165a9       Config memory filled

0x165aa       Worker setconfig error
0x165ab       Worker pipe not created
0x17005       Worker ffst config not okay
0x165ac       Worker mutexsem not released
0x17010       Worker already active
```

```
0x17015       Worker initworker failed
0x17050       Worker not active
```

FFSTSetConfiguration - Remarks

The library FFST.LIB must be linked with object files that use FFSTSetConfiguration

The *packet_revision_number* parameter defines if pointers point to ASCII or UniCode character data.

FFSTSetConfiguration - Related Functions

- FFSTQueryConfiguration

FFSTSetConfiguration - Example Code

The following example will instruct FFST to stop wrapping dumps. The rest of the parameters are left to their earlier values.

```
#define INCL_FFST
#include <unichar.h>
#include <os2.h>
APIRET rc;

CONFIGPARMS  FFSTConfig;
PCONFIGPARMS  pFFSTConfig  *FFSTConfig;

FFSTConfig.dump_file_wrap = FFST_DUMP_WRAP_OFF; /* Set dump wrap to off*/
/* set rest of the parameters to indicate no change */

FFSTConfig.dump_file_directory_size = FFST_DUMP_FILE_DIRECTORY_SIZE_NO_CHANGE;
FFSTConfig.keep_dup_dump = FFST_KEEP_DUP_DUMP_NO_CHANGE;
FFSTConfig.dump_file_directory_length = FFST_DUMP_FILE_DIERCTORY_LENGTH_NO_CHANGE;
FFSTConfig.no_of_disabled_products = FFST_NO_OF_PROBE_DISABLED_PRODUCTS_NO_CHANGE;
FFSTConfig.dump_file_directory = NULL;
FFSTConfig.PProductData = NULL;

rc = FFSTSetConfiguration(  pFFSTConfig);
if (rc |= 0)                                    /* If Problem   */
          {                                     /* reason       */
          printf("FFSTConfigure error: return code = %d",rc);
          return;
          }
```

FFSTSetConfiguration - Topics

Select an item:
Syntax

LogChangeEventFilter

LogChangeEventFilter - Syntax

LogChangeEventFilter is used to alter the filter that is associated with an event-notification request. In addition to changing the filter, you can specify current entries that are to be purged before the filter change takes effect.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pChangeEventFilter;
APIRET    rc;

rc = LogChangeEventFilter(service, pChangeEventFilter);
```

LogChangeEventFilter Parameter - service

service (ULONG) - input
The class of Logging Service:

1	Error logging

All other values are reserved for future use.

LogChangeEventFilter Parameter - pChangeEventFilter

pChangeEventFilter (PVOID) - in/out
A pointer to the LogChangeEventFilter parameter packet.

For Error Logging, this is a pointer to a LCEFREQUEST structure.

LogChangeEventFilter Return Value - rc

rc (APIRET) - returns
Return code.

LogChangeEventFilter returns the following values:

- 0 No error
- 523 Error If invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1751 Invalid LF flag
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2503 RAS notif entry not found
- 2504 RAS notif entry deleted
- 2505 RAS entry filter unchanged

LogChangeEventFilter - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pChangeEventFilter (PVOID) - input/output
A pointer to the LogChangeEventFilter parameter packet.

For Error Logging, this is a pointer to a LCEFREQUEST structure.

rc (APIRET) - returns
Return code.

LogChangeEventFilter returns the following values:

- 0 No error
- 523 Error If invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1751 Invalid LF flag
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2503 RAS notif entry not found
- 2504 RAS notif entry deleted
- 2505 RAS entry filter unchanged

LogChangeEventFilter - Remarks

The library LFAPI.LIB must be linked with object files that use LogChangeEventFilter

---

LogChangeEventFilter - Related Functions

- LogOpenEventNotification
- LogCloseEventNotification
- LogWaitEvent
- LogReadEntry

LogChangeEventFilter - Example Code

The following example changes the event-notification filter for an event notification to a NULL filter (that is, any Error Log entry that is logged will cause an event notification to be sent). The sample will also purge any event notifications that might be pending at the time the LogChangeEventFilter call is made.

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>

{
APIRET rc;                                      /* return code */
ULONG service;
LCEFREQUEST change_event_filter_packet;
HLOGNOTIFY log_notify;

service =  ERROR_LOGGING_SERVICE;

/* Construct the LogChangeEventFilter parameter packet */
change_event_filter_packet.packet_size = sizeof(LCEFREQUEST);
change_event_filter_packet.packet_revision_number = LF_UNI_API;
change_event_filter_packet.purge_flags = PURGE_EVENT_NOTIFICATION;
change_event_filter_packet.LogNotify = log_notify;
change_event_filter_packet.pFilter = NULL;

rc = LogChangeEventFilter(service,              /* service */
                &change_event_filter_packet) /* parameter packet*/
if (rc |= 0)
   {
    printf("LogChangeEventFilter error: return code = %d",rc);
    return;
   }
```

LogChangeEventFilter - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions

LogCloseEventNotification

LogCloseEventNotification - Syntax

LogCloseEventNotification closes event-notification requests.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pCloseEventNotification;
APIRET    rc;

rc = LogCloseEventNotification(service, pCloseEventNotification);
```

LogCloseEventNotification Parameter - service

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

LogCloseEventNotification Parameter - pCloseEventNotification

pCloseEventNotification (PVOID) - in/out
A pointer to the LogCloseEventNotification parameter packet.

For Error Logging, this is a pointer to a LCENREQUEST structure.

LogCloseEventNotification Return Value - rc

rc (APIRET) - returns
Return code.

LogCloseEventNotification returns the following values:

- 0 No error

- 523 Error lf invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2502 RAS internal memory failure
- 2503 RAS notif entry not found

LogCloseEventNotification - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pCloseEventNotification (PVOID) - in/out
A pointer to the LogCloseEventNotification parameter packet.

For Error Logging, this is a pointer to a LCENREQUEST structure.

rc (APIRET) - returns
Return code.

LogCloseEventNotification returns the following values:

- 0 No error
- 523 Error lf invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2502 RAS internal memory failure
- 2503 RAS notify entry not found

LogCloseEventNotification - Remarks

The library LFAPI.LIB must be linked with object files that use LogCloseEventNotification

---

LogCloseEventNotification - Related Functions

- LogOpenEventNotification
- LogChangeEventFilter
- LogWaitEvent
- LogReadEntry

LogCloseEventNotification - Example Code

The following example closes an event-notification mechanism that is connected to the Error Logging service.

```
  #define INCL_LOGGING
  #include <unidef.h>
  #include <os2.h>
  #include <stdio.h>
  #include <lfdef.h>

  {
   APIRET rc;                           /* return code */
   ULONG service;
   LCENREQUEST close_event_packet;
   HLOGNOTIFY       log_notify;

   service =       ERROR_LOGGING_SERVICE;

   /* Construct the LogChangeEventFilter parameter packet  */
   close_event_packet.packet_size = sizeof(LCENREQUEST);
   close_event_packet.packet_revision_number = LF_UNI_API;
   close_event_packet.LogNotify = log_notify;
   rc = LogCloseEventNotification(service,        /* service */
                &close_event_packet)        /* parameter packet */
   if (rc |= 0)
      {
       printf("LogCloseEventNotification error: return code = %d",rc);
       return;
      }
```

LogCloseEventNotification - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions

LogCloseFile

LogCloseFile - Syntax

LogCloseFile closes a file that was previously opened by LogOpenFile.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pCloseFile;
```

```
APIRET    rc;

rc = LogCloseFile(service, pCloseFile);
```

LogCloseFile Parameter - service

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

LogCloseFile Parameter - pCloseFile

pCloseFile (PVOID) - input
A pointer to the LogCloseFile parameter packet.

For Error Logging, this is a pointer to a LCFREQUEST structure.

LogCloseFile Return Value - rc

rc (APIRET) - returns
Return code.

LogCloseFile returns one of the following values:

- 0 No error
- 520 Error LF buf too small
- 5 Error access denied
- 523 Error LF invalid service
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1703 Invalid data pointer
- 1706 Invalid LF parm packet ptr
- 1761 Error LF invalid packet size

LogCloseFile - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pCloseFile (PVOID) - input
A pointer to the LogCloseFile parameter packet.

For Error Logging, this is a pointer to a LCFREQUEST structure.

rc (APIRET) - returns
Return code.

LogCloseFile returns one of the following values:

- 0 No error
- 520 Error LF buf too small
- 5 Error access denied
- 523 Error LF invalid service
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1703 Invalid data pointer
- 1706 Invalid LF parm packet ptr
- 1761 Error LF invalid packet size

LogCloseFile - Remarks

The library LFAPI.LIB must be linked with object files that use LogCloseFile

---

LogCloseFile - Related Functions

- LogOpenFile
- LogReadEntry

LogCloseFile - Example Code

The following example closes a log file (log_file_ID = 2) that was opened with LogOpenFile.

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>

{
APIRET rc;                          /* return code */
ULONG service;
LCFREQUEST close_file_packet;

service = ERROR_LOGGING_SERVICE;

/* Construct the LogOpenFile parameter packet  */
close_file_packet.packet_size = sizeof(LCFREQUEST);
close_file_packet.packet_revision_number = LF_UNI_API;
close_file_packet.log_file_ID = 2;
```

```
  rc = LogCloseFile(service,                              /* service */
                    &close_file_packet)           /* parameter packet  */
  if (rc |= 0)
  {
   printf("LogCloseFile error: return code = %d",rc);
   return;
  }
```

LogCloseFile - Topics

Select an item:

LogFormatEntry

LogFormatEntry - Syntax

LogFormatEntry formats a Log Entry for display.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pFormatEntry;
APIRET    rc;

rc = LogFormatEntry(service, pFormatEntry);
```

LogFormatEntry Parameter - service

service (ULONG) - input
The class of Logging Service:

1	Error logging

All other values are reserved for future use.

LogFormatEntry Parameter - pFormatEntry

pFormatEntry (PVOID) - input
A pointer to the LogFormatEntry parameter packet.

For Error Logging, this is a pointer to a LFEREQUEST structure.

LogFormatEntry Return Value - rc

rc (APIRET) - returns
Return code.

LogFormatEntry returns the following values:

- 0 No error
- 2 Error file not found
- 520 Error LF buf too small
- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 Invalid LF parm packet ptr
- 1751 Invalid LF flag
- 1761 Error LF invalid packet size
- 1770 Invalid log entry record
- 1771 No log entry format template available
- 2507 RAS unicode conversion error
- 2600 RAS invalid locale object

LogFormatEntry - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pFormatEntry (PVOID) - input
A pointer to the LogFormatEntry parameter packet.

For Error Logging, this is a pointer to a LFEREQUEST structure.

rc (APIRET) - returns
Return code.

LogFormatEntry returns the following values:

- 0 No error
- 2 Error file not found
- 520 Error LF buf too small

- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 Invalid LF parm packet ptr
- 1751 Invalid LF flag
- 1761 Error LF invalid packet size
- 1770 Invalid log entry record
- 1771 No log entry format template available
- 2507 RAS unicode conversion error
- 2600 RAS invalid locale object

LogFormatEntry - Remarks

ADDITIONAL RETURNS INFORMATION

The data will be passed back in multiple occurrences of the following LTD (Length, Type, Data) format:

ULONG	length
ULONG	type
UniChar	data[n]
or	
Char	data[n]

Where:

length (ULONG) is the length, in bytes, of this detail record (includes length, type, and data fields).

type (ULONG) is an integer value that represents the type of data being passed back.

An *error record* is created by the system when an error in a system or application program triggers a probe in that program. Error records contain detailed information to help you diagnose the error. Error records are also called *DET1* records. Records created by a back level logging system are called *DET4* records.

A *control record* is created by the system when you make changes to the way errors are logged. For example, when you suspend error logging or direct error logging to a new file, the system records that change in a control record. Control records are also called *DET2* records. Control records are new for FFST technology and are not available in records created by a back level logging system.

The following are the current Error Logging type values and their meanings:

Type	Meaning	From Record Type
0005	Date heading	DET1, DET2, or DET4 record
0006	Date	DET1, DET2, or DET4 record
0007	Time heading	DET1, DET2, or DET4 record
0008	Time	DET1, DET2, or DET4

		record
0009	Entry ID heading	DET1, DET2, or DET4 record
0010	Entry ID	DET1, DET2, or DET4 record
0011	Severity heading	DET1 record
0012	Severity	DET1 record
0013	Module name heading	DET1 record
0014	Module name	DET1 record
0015	Directory name heading	DET1 record
0016	Directory name	DET1 record
0017	Error message heading	DET1 record
0018	Error message text	DET1 record
0019	Probe ID heading	DET1 record
0020	Probe ID text	DET1 record
0021	Probe Flags heading	DET1 record
0022	Probe Flags	DET1 record
0023	Template Repository pathname heading	DET1 record
0024	Template Repository pathname text	DET1 record
0025	Template ID heading	DET1 record
0026	Template ID text	DET1 record
0027	Dump generated heading	DET1 record
0028	Dump Generated text	DET1 record
0029	Trace file generated heading	DET1 record
0030	Trace File generated text	DET1 record
0031	Process dump generated heading	DET1 record
0032	Process Dump generated text	DET1 record
0040	Failure Causes heading	DET1

		record
0041	Failure Cause (Could be 4 of these)	DET1 record
0050	Failure Actions heading	DET1 record
0051	Failure Action (Could be 4 of these)	DET1 record
0060	Install Causes heading	DET1 record
0061	Install Cause (Could be 4 of these)	DET1 record
0070	Install Actions heading	DET1 record
0071	Install Action (Could be 4 of these)	DET1 record
0080	User Causes heading	DET1 record
0081	User Cause (Could be 4 of these)	DET1 record
0090	User Actions heading	DET1 record
0091	User Action (Could be 4 of these)	DET1 record
0100	Return Code heading	DET1 record
0101	Return Code text	DET1 record
0110	Dump File name heading	DET1 record
0111	Dump File name text	DET1 record
0112	Dump formatter heading	DET1 record
0113	Dump Formatter text	DET1 record
0114	Dump File Deletion Date heading	DET1 record
0115	Dump File Deletion Date	DET1 record
0116	Dump File Deletion Time heading	DET1 record
0117	Dump File Deletion Time	DET1 record
0120	Trace File name heading	DET1 record
0121	Trace File name text	DET1 record
0122	Trace formatter heading	DET1 record
0123	Trace formatter text	DET1 record
0124	Trace File Deletion Date heading	DET1 record

0125	Trace File Deletion Date	DET1 record
0126	Trace File Deletion Time heading	DET1 record
0127	Trace File Deletion Time	DET1 record
0130	Process Dump File name heading	DET1 record
0131	Process Dump File name text	DET1 record
0132	Process Dump formatter heading	DET1 record
0133	Process Dump formatter text	DET1 record
0134	Process Dump Deletion Date heading	DET1 record
0135	Process Dump File Deletion Date	DET1 record
0136	Process Dump Deletion Time heading	DET1 record
0137	Process Dump File Deletion Time	DET1 record
0140	PCT heading	DET1 record
0141	PCT Execution Parameters	DET1 record
0150	DMI vendor tag heading	DET1 record
0151	DMI vendor tag text	DET1 record
0155	DMI tag heading	DET1 record
0156	DMI tag text	DET1 record
0165	DMI product ID heading	DET1 record
0166	DMI product ID text	DET1 record
0170	DMI revision heading	DET1 record
0171	DMI revision text	DET1 record
0172	DMI modification level heading	DET1 record
0173	DMI modification level text	DET1 record
0174	DMI fix level heading	DET1 record
0175	DMI fix level text	DET1 record
0195	Machine type heading	DET1 record

0196	Machine type text	DET1 record
0200	Machine serial number heading	DET1 record
0201	Machine serial number text	DET1 record
0205	Hostname heading	DET1 record
0206	Hostname text	DET1 record
0210	User Data heading	DET1 record
0211	User data text	DET1 record
0213	Action heading	DET2 record
0214	Action text	DET2 record
0215	Old Value heading	DET2 record
0216	Old Value text	DET2 record
0220	New Value heading	DET2 record
0221	New Value text	DET2 record
0222	Created by backlevel text	DET4 record
0223	Record ID heading	DET4 record
0224	Record ID	DET4 record
0225	Qualifier heading	DET4 record
0226	Qualifier	DET4 record
0227	Originator heading	DET4 record
0228	Originator	DET4 record
0229	User data	DET4 record
0230	Process name heading	DET4 record
0231	Process name	DET4 record
0232	FMTDLL heading	DET4 record
0233	FMTDLL name	DET4 record
0234	FMTDLL text	DET4 record
0235	GA component ID heading	DET4

```
                                    record

0236       GA component ID          DET4
                                    record

0237       GA release level heading DET4
                                    record

0238       GA release level         DET4
                                    record

0239       GA software name heading DET4
                                    record

0240       GA software name         DET4
                                    record

0241       Generic alert subvector  DET4
           heading                  record

0242       Generic alert subvector  DET4
           text                     record

0243       Probable causes subvector DET4
           heading                  record

0244       Probable causes subvector DET4
           text                     record

0245       User causes subvector    DET4
           heading                  record

0246       Install causes subvector DET4
           heading                  record

0247       Failure causes subvector DET4
           heading                  record

0248       Subvector key heading    DET4
                                    record

0249       Subvector key type       DET4
                                    record

0250       Subvector key text       DET4
                                    record

0251       Hex dump text            DET4
                                    record
```

data (UniChar[]) or **data (Char[])** is a variable length area that contains the formatted data.

Note: If there is no data, only the length and type portion of the record will be returned. The length would indicate that there is no data.

The library LFAPI.LIB must be linked with object files that use LogFormatEntry

The *packet_revision_number* parameter defines if pointers point to ASCII or UniCode character data.

LogFormatEntry - Related Functions

- LogReadEntry

LogFormatEntry - Example Code

The following example formats an Error Log record for display. The calling program has placed the address of the locale object in */locale* .

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>

{
APIRET rc;                              /* return code */
ULONG service;
LFEREQUEST format_entry_packet;
BYTE log_entry_buffer[1024];
UniChar string_buffer[4096];
#define STRING_BUFFER_LENGTH 4096
ULONG string_buffer_length;
LocaleObject locale;

service = ERROR_LOGGING_SERVICE;
string_buffer_length = STRING_BUFFER_LENGTH;
rc = UniCreateLocaleObject(UNLUCS_STRING_POINTNER,(UniChar *) L"",&locale);
if (rc |= 0)
  {
   printf("UniCreateLocaleObject error: return code = %d",rc);
   return;
  }

/* Construct the Error Log Service format packet  */
format_entry.packet.packet_size = sizeof(LFEREQUEST);
format_entry_packet.packet_revision_number = WPOS_RELEASE_1;
format_entry_packet.log_entry_buffer =  &log_entry_buffer;
format_entry_packet.flags = ERR_FORMAT_DETAIL_DATA;
format_entry_packet.locale_object = locale;
format_entry_packet.string_buffer_length = &string_buffer_length;
format_entry_packet.string_buffer =  &string_buffer;
rc = LogFormatEntry(service,            /* service */
          &format_entry_packet)   /* parameter packet */
if (rc |= 0)
  {
   printf("LogFormatEntry error: return code = %d",rc);
   return;
  }
```

LogFormatEntry - Topics

Select an item:

LogOpenEventNotification

LogOpenEventNotification - Syntax

LogOpenEventNotification registers a consumer with the Logging Service, so that the consumer will receive notification when specific log records have been created. Consumers specify which log records they will be notified about by providing filtering information. If no filter data structure is provided, all events that are logged to the specified log file will cause event notifications to be forwarded to the consumer. LogOpenEventNotification returns an ID used to reference this notification request.

Notifications will be sent for only those records placed in the error log by the FFSTProbe API.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG           service;
PVOID           pOpenEventNotification;
APIRET          rc;

rc = LogOpenEventNotification(service, pOpenEventNotification);
```

LogOpenEventNotification Parameter - service

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

LogOpenEventNotification Parameter - pOpenEventNotification

pOpenEventNotification (PVOID) - input/output
A pointer to the LogOpenEventNotification parameter packet.

For Error Logging, this is a pointer to a LOENREQUEST structure.

LogOpenEventNotification Return Value - rc

rc (APIRET) - returns
Return code.

LogOpenEventNotification returns the following values:

- 0 No error
- 523 Error LF invalid service

- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 RAS invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1751 RAS invalid flag
- 1757 RAS invalid log notify ptr
- 1761 RAS invalid packet size

LogOpenEventNotification - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pOpenEventNotification (PVOID) - in/out
A pointer to the LogOpenEventNotification parameter packet.

For Error Logging, this is a pointer to a LOENREQUEST structure.

rc (APIRET) - returns
Return code.

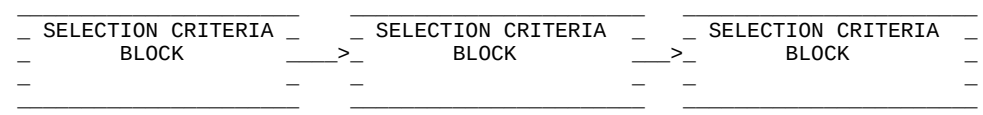
LogOpenEventNotification returns the following values:

- 0 No error
- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 RAS invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 RAS invalid parm packet ptr
- 1751 RAS invalid flag
- 1757 RAS invalid log notify ptr
- 1761 RAS invalid packet size

LogOpenEventNotification - Remarks

The event-notification filter is a flexible data structure that is used to specify the class of events whose notifications will be received through the event-notification mechanism. It is available to event consumers (through LogOpenEventNotification and LogChangeEventFilter) and to log file readers (through LogReadEntry). This provides a common search criteria when waiting for events and reading selected entries within a log file.

EVENT NOTIFICATION FILTER STRUCTURE



The event-notification filter consists of an array of one or more selection criteria blocks. Each selection criteria block contains a small header block that specifies the revision of the filter and points to the next selection criteria block.

Each selection criteria block consists of an array of selection criteria subblocks. Each selection criteria subblock contains three pieces of

information:

1. The ID of an attribute that is contained within this class of log entry.

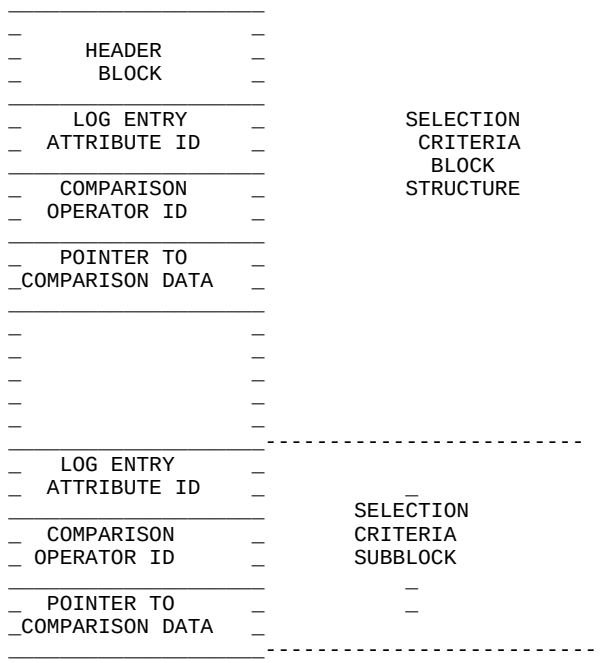
   For more information see the entry_attribute_ID parameter.

2. A comparison operator that is to be applied against the specified log entry attribute.

   For more information see the comparison_operator_ID parameter.

3. A pointer to a data that can be either ASCII or UniCode data. The data type is determined by the packet_revision_number

The following diagram summarizes the structure of a selection criteria block:

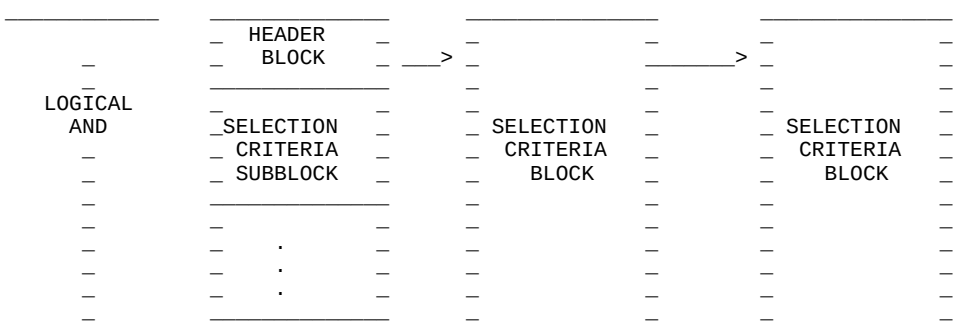


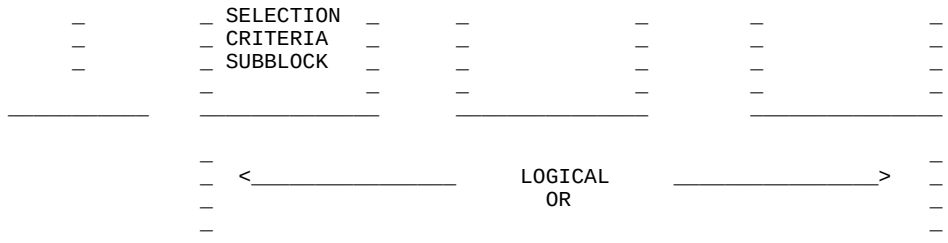
Each selection criteria subblock specifies a comparison with respect to an attribute within a log entry. The result of the comparison is a Boolean value. The Boolean value of the array of selection criteria subblocks (within a selection criteria block) is the logical AND of the Boolean values for each subblock.

If an event-notification filter contains more than one selection criteria block, the entire chain of selection criteria blocks is considered to resolve to the logical OR of the Boolean values of the individual blocks. In this manner, a consumer can construct appropriately complex event-discrimination filters.

The following diagram illustrates the logical representation of an event-notification filter:

EVENT NOTIFICATION FILTER STRUCTURE





The library LFAPI.LIB must be linked with object files that use LogOpenEventNotification

---

LogOpenEventNotification - Related Functions

- LogChangeEventFilter
- LogWaitEvent
- LogCloseEventNotification
- LogReadEntry

LogOpenEventNotification - Example Code

The following example opens error-logging event notification. It will initialize an event-notification filter that specifies any Error Log entry that has a product manufacturer named "IBM" and a severity less than 4.

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>
#define ERROR_LOG_FILE_ID 1

ULONG service;
ULONG severity = 4;
ULONG entry_id = 12;
LOENREQUEST open_event_packet;
HLOGNOTIFY log_notify;
SUBBLOCK subblock1, subblock2, subblock3;
HEADERBLOCK headerblock1, headerblock2;
FILTERBLOCK filter;

UniChar *manufacturer_name = L"IBM";

service = ERROR_LOGGING_SERVICE;

/*  Construct an event notification filter with 2 header blocks. */
/*  The first header block points to a single subblock.          */
/*  The second header block points to a chain of two subblocks.  */

filter.packet_size = sizeof(FILTERBLOCK);
filter.packet_revision_number = LF_UNI_API;
filter.header_block = &headerblock1;

/*-------------construct headerblock1---------------------*/
headerblock1.pSubblock = &subblock1;
headerblock1.pNextBlock = &headerblock2;

/*construct subblock1 of headerblock1*/
subblock1.entry_attribute_ID = LOG_ERROR_DMI_VENDOR_TAG;
subblock1.comparison_operator = LOG_ERROR_EQUAL;
subblock1.comparison_data_ptr = &manufacturer_name;
subblock1.next_subblock = NULL;
```

```
/*------------construct headerblock2----------------------*/
headerblock2.pSubblock = &subblock2;
headerblock2.pNextBlock = NULL;

/*construct subblock2 of headerblock2*/
subblock2.entry_attribute_ID = LOG_ERROR_SEVERITY;
subblock2.comparison_operator = LOG_ERROR_LESS_THAN;
subblock2.comparison_data_ptr = severity;
subblock2.comparison_data_length = sizeof(ULONG);
subblock2.next_subblock = &subblock3;

/*construct subblock3 of headerblock2*/
subblock3.entry_attribute_ID = LOG_ERROR_ENTRY_ID;
subblock3.comparison_operator = LOG_ERROR_GREATER_THAN;
subblock3.comparison_data_ptr = entry_id;
subblock3.comparison_data_length = sizeof(ULONG);
subblock3.next_subblock = null;

/* Construct the LogOpenEventNotification parameter packet.*/
open_event_packet.packet_size = sizeof(LOENREQUEST);
open_event_packet.packet_revision_number = LF_UNI_API;
open_event_packet.log_file_ID = ERROR_LOG_FILE_ID;
open_event_packet.pLogNotify = &log_notify;
open_event_packet.pFilter = &filter;
open_event_packet.read_flags = 0;

rc = LogOpenEventNotification(service,        /*service*/
                    &open_event_packet); /*parameter packet*/

if (rc |= 0)
  {
   printf("LogOpenEventNotification error: return code = %d",rc);
   return;
  }
```

LogOpenEventNotification - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions

LogOpenFile

LogOpenFile - Syntax

LogOpenFile opens a log file so that it can be read using LogReadEntry. LogOpenFile returns a log_file_ID number that is used to refer to the file.

For Error Logging:

Specify either a log_file_ID number or a path name. If the log_file_ID number is specified and the log_file_ID is valid the path name is returned. If a path name is specified, the Logging Service will return the log_file_ID that corresponds to the file. (The file will be opened if it is not already open.)

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pOpenFile;
APIRET    rc;

rc = LogOpenFile(service, pOpenFile);
```

LogOpenFile Parameter - service

service (ULONG) - input
The class of Logging Service:

1	Error logging

All other values are reserved for future use.

LogOpenFile Parameter - pOpenFile

pOpenFile (PVOID) - in/out
A pointer to the LogOpenFile parameter packet.

For Error Logging, this is a pointer to a LOFREQUEST structure.

LogOpenFile Return Value - rc

rc (APIRET) - returns
Return code.

LogOpenFile returns the following values:

- 0 No error
- 2 Error file not found
- 99 Error device in use
- 108 Error drive locked
- 110 Error open failed
- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1704 Invalid LF filename length
- 1705 Invalid LF filename ptr

- 1706 Invalid LF parm packet ptr
- 1732 Invalid LF log file
- 1761 Error LF invalid packet size

LogOpenFile - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error logging

All other values are reserved for future use.

pOpenFile (PVOID) - in/out
A pointer to the LogOpenFile parameter packet.

For Error Logging, this is a pointer to a LOFREQUEST structure.

rc (APIRET) - returns
Return code.

LogOpenFile returns the following values:

- 0 No error
- 2 Error file not found
- 99 Error device in use
- 108 Error drive locked
- 110 Error open failed
- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1704 Invalid LF filename length
- 1705 Invalid LF filename ptr
- 1706 Invalid LF parm packet ptr
- 1732 Invalid LF log file
- 1761 Error LF invalid packet size

LogOpenFile - Remarks

The library LFAPI.LIB must be linked with object files that use LogOpenFile

---

LogOpenFile - Related Functions

- LogCloseFile
- LogReadEntry

LogOpenFile - Example Code

The following example verifies that the default file (1) is opened.

```
  #define INCL_LOGGING
  #include <unidef.h>
  #include <os2.h>
  #include <stdio.h>
  #include <lfdef.h>
  #define ERROR_LOG_FILE_ID 1

  {
  APIRET rc;                      /* return code */
  ULONG service;
  UniChar filenameø256!;
  ULONG filename_length;
  ULONG log_file_ID;
  LOFREQUEST open_file_packet;

  service = ERROR_LOGGING_SERVICE;

  /* Construct the LogOpenFile parameter packet  */
  open_file_packet.packet_size = sizeof(LOFREQUEST);
  open_file_packet.packet_revision_number = LF_UNI_API;
  log_file_ID = ERROR_LOG_FILE_ID;
  open_file_packet.log_file_ID = &log_file_ID;
  open_file_packet.filename_length = &filename_length; /*Indicates use the
                                                         Log File ID   */
  open_file_packet.filename = &filename;

  rc = LogOpenFile(service,                     /* service */
              &open_file_packet)             /* parameter packet  */

  if (rc |= 0)
    {
     printf("LogOpenFile error: return code = %d",rc);
     return;
    }
```

*

LogOpenFile - Topics

Select an item:

LogReadEntry

LogReadEntry - Syntax

LogReadEntry reads a specified log entry from a log file.

You can specify a filter that is used to select only entries of a desired class. The format of the filter is described by the event-filter data structure FILTERBLOCK.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pReadEntry;
APIRET    rc;

rc = LogReadEntry(service, pReadEntry);
```

LogReadEntry Parameter - service

service (ULONG) - input
The class of Logging Service:

1 Error Logging

All other values are reserved for future use.

LogReadEntry Parameter - pReadEntry

pReadEntry (PVOID) - in/out
A pointer to the LogReadEntry parameter packet.

For Error Logging, this is a pointer to a LREREQUEST structure.

The caller can specify an event filter that is used to select only entries of a desired class. The format of the event filter is the same as the event notification filter that is described in the LogOpenEventNotification API. The caller also provides an entry key data structure. Entry keys are generated by this API and by the LogWaitEvent API.

Log File searching typically begins at the entry that follows the one that is specified within the event key. The event key is updated as new entries are read. If no event filter is provided, the LogReadEntry API will read the entry that is pointed to by the event key. The caller also has the option of starting a search at the logical beginning of the Log File.

LogReadEntry Return Value - rc

rc (APIRET) - returns
Return code.

LogReadEntry returns the following values:

- 0 No error
- 520 Error LF buf too small
- 523 Error LF invalid service

- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 Invalid LF parm packet ptr
- 1750 Invalid LF at end of log
- 1751 Invalid LF flag
- 1759 Invalid LF entry key
- 1761 Error LF invalid packet size

LogReadEntry - Parameters

service (ULONG) - input
The class of Logging Service:

1 Error Logging

All other values are reserved for future use.

pReadEntry (PVOID) - in/out
A pointer to the LogReadEntry parameter packet.

For Error Logging, this is a pointer to a LREREQUEST structure.

The caller can specify a filter that is used to select only entries of a desired class. The format of the filter is the same as the filter that is described in the *LogOpenEventNotification* API pFilter parameter. The caller also provides an entry key data structure. Entry keys are generated by this API and by the LogWaitEvent API.

Log File searching typically begins at the entry that follows the one that is specified within the event key. The event key is updated as new entries are read. If no event filter is provided, the LogReadEntry API will read the entry that is pointed to by the event key. The caller also has the option of starting a search at the logical beginning of the Log File.

rc (APIRET) - returns
Return code.

LogReadEntry returns the following values:

- 0 No error
- 520 Error LF buf too small
- 523 Error LF invalid service
- 524 Error LF general failure
- 1703 Invalid data pointer
- 1701 Invalid LF log file id
- 1702 Invalid LF packet revision number
- 1706 Invalid LF parm packet ptr
- 1750 Invalid LF at end of log
- 1751 Invalid LF flag
- 1759 Invalid LF entry key
- 1761 Error LF invalid packet size

LogReadEntry - Remarks

The library LFAPI.LIB must be linked with object files that use LogReadEntry.

LogReadEntry - Related Functions

- LogFormatEntry
- LogOpenEventNotification
- LogWaitEvent

LogReadEntry - Example Code

The following example reads an entry from the default Error Logging log file. It searches for the first (that is, most recent) entry in the file that has a product manufacturer named "IBM" and a severity less than 4.

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>
#define ERROR_LOG_FILE_ID 1
#define START_AT_FIRST_ENTRY 0

{
ULONG service;
ULONG severity = 4;
ULONG entry_id = 12;
LREREQUEST read_entry_packet;
EVENTKEY EventKey;
BYTE log_entry_bufferø2048!;
ULONG log_entry_buffer_length;
SUBBLOCK subblock1, subblock2, subblock3;
HEADERBLOCK headerblock1, headerblock2;
FILTERBLOCK filter;

UniChar *manufacturer_name = L"IBM";

service = ERROR_LOGGING_SERVICE;

/*  Construct an event notification filter with 2 header blocks.    */
/*  The first header block points to a single subblock.      */
/*  The second header block points to a chain of two subblocks.  */

filter.packet_size = sizeof(FILTERBLOCK);
filter.packet_revision_number = LF_UNI_API;
filter.header_block = &headerblock1;

/*-------------construct headerblock1---------------------*/
headerblock1.pSubblock = &subblock1;
headerblock1.pNextBlock = &headerblock2;

/*construct subblock1 of headerblock1*/
subblock1.entry_attribute_ID = LOG_ERROR_DMI_VENDOR_TAG;
subblock1.comparison_operator = LOG_ERROR_EQUAL;
subblock1.comparison_data_ptr = &manufacturer_name;
subblock1.next_subblock = NULL;

/*------------construct headerblock2----------------------*/
headerblock2.pSubblock = &subblock2;
headerblock2.pNextBlock = NULL;

/*construct subblock2 of headerblock2*/
subblock2.entry_attribute_ID = LOG_ERROR_SEVERITY;
subblock2.comparison_operator = LOG_ERROR_LESS_THAN;
subblock2.comparison_data_ptr = severity;
subblock2.comparison_data_length = sizeof(ULONG);
subblock2.next_subblock = &subblock3;

/*construct subblock3 of headerblock2*/
subblock3.entry_attribute_ID = LOG_ERROR_ENTRY_ID;
subblock3.comparison_operator = LOG_ERROR_GREATER_THAN;
subblock3.comparison_data_ptr = entry_id;
subblock3.comparison_data_length = sizeof(ULONG);
subblock3.next_subblock = null;
```

```
/* Construct the LogReadEntry parameter packet. */
read_entry_packet.packet_size = sizeof(LREREQUEST);
read_entry_packet.packet_revision_number = LF_UNI_API;
read_entry_packet.log_file_ID = ERROR_LOG_FILE_ID;
read_entry_packet.flags = START_AT_FIRST_ENTRY;
read_entry_packet.pEventKey = &EventKey;
read_entry_packet.pFilter = &filter;
log_entry_buffer_length = sizeof(log_entry_buffer);
read_entry_packet.pLogEntryBufferLength = &log_entry_buffer_length;
read_entry_packet.LogEntryBuffer = &log_entry_buffer;

rc = LogReadEntry(service,           /*service*/
            &read_entry_packet); /*parameter packet*/

if (rc |= 0)
  {
   printf("LogReadEntry error: return code = %d",rc);
   return;
  }
```

LogReadEntry - Topics

Select an item:
Syntax
Parameters
Returns
Remarks
Example Code
Related Functions

LogWaitEvent

LogWaitEvent - Syntax

LogWaitEvent waits for event-notification information for which the process registered. You receive the event-key data structure and log entry data from the Logging Service.

For details on event-notification filters, see LogOpenEventNotification.

```
#define INCL_LOGGING
#include <os2.h>
#include <lfdef.h>

ULONG     service;
PVOID     pWaitEvent;
APIRET    rc;

rc = LogWaitEvent(service, pWaitEvent);
```

LogWaitEvent Parameter - service

service (ULONG) - input
The class of Logging Service:

1	Error Logging

All other values are reserved for future use.

LogWaitEvent Parameter - pWaitEvent

pWaitEvent (PVOID) - in/out
A pointer to the LogWaitEvent parameter packet.

For Error Logging, this is a pointer to a LWEREQUEST structure.

LogWaitEvent Return Value - rc

rc (APIRET) - returns
Return code.

LogWaitEvent returns the following values:

- 0 No error
- 520 Error LF buf too small
- 523 Error LF invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1704 Invalid LF filename length
- 1705 Invalid LF filename ptr
- 1706 Invalid LF parm packet ptr
- 1709 Invalid logrecord buffer ptr
- 1751 Invalid LF flag
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2055 Error LF timeout
- 2508 RAS invalid eventkey ptr
- 2509 RAS invalid pathlen ptr
- 2510 RAS invalid buflen ptr

LogWaitEvent - Parameters

service (ULONG) - input
The class of Logging Service:

1	Error Logging

All other values are reserved for future use.

pWaitEvent (PVOID) - in/out
A pointer to the LogWaitEvent parameter packet.

For Error Logging, this is a pointer to a LWEREQUEST structure.

rc (APIRET) - returns
Return code.

LogWaitEvent returns the following values:

- 0 No error
- 520 Error LF buf too small
- 523 Error LF invalid service
- 1703 Invalid data pointer
- 1702 Invalid LF packet revision number
- 1704 Invalid LF filename length
- 1705 Invalid LF filename ptr
- 1706 Invalid LF parm packet ptr
- 1709 Invalid logrecord buffer ptr
- 1751 Invalid LF flag
- 1758 RAS invalid log notify id
- 1761 RAS invalid packet size
- 2055 Error LF timeout
- 2508 RAS invalid eventkey ptr
- 2509 RAS invalid pathlen ptr
- 2510 RAS invalid buflen ptr

LogWaitEvent - Remarks

The library LFAPI.LIB must be linked with object files that use LogWaitEvent.

LogWaitEvent - Related Functions

- LogOpenEventNotification
- LogCloseEventNotification
- LogChangeEventFilter
- LogReadEntry

LogWaitEvent - Example Code

The following example waits for an event notification from the Error Logging service. The call will wait indefinitely for an event notification.

```
#define INCL_LOGGING
#include <unidef.h>
#include <os2.h>
#include <stdio.h>
#include <lfdef.h>

{
APIRET rc;              /* return code */
ULONG service;
LWEREQUEST log_wait_event_packet;
```

```
HLOGNOTIFY log_notify;
EVENTKEY EventKey;
BYTE log_entry_bufferø4096!;
UniChar pathnameø512!;
ULONG pathname_length = sizeof(pathname);


service =  ERROR_LOGGING_SERVICE;

/* Construct the LogChangeEventFilter parameter packet  */
log_wait_event_packet.packet_size = sizeof(LWEREQUEST);
log_wait_event_packet.packet_revision_number = LF_UNI_API;
log_wait_event_packet.LogNotify = log_notify;
log_wait_event_packet.pEventKey = &EventKey;
log_wait_event_packet.pLogEntryBuffer = &log_entry_buffer;
log_wait_event_packet.timeout = 0;
log_wait_event_packet.queue_flags = 0;
log_wait_event_packet.pathname_length = &pathname_length;
log_wait_event_packet.pathname = pathname;

rc = LogWaitEvent(service,                      /* service */
                  &log_wait_event_packet)       /* parameter packet */
if (rc |= 0)
  {
   printf("LogWaitEvent error: return code = %d",rc);
   return;
  }
```

LogWaitEvent - Topics

Select an item:

TraceCreateEntry

TraceCreateEntry - Syntax

TraceCreateEntry creates an event entry in the system event buffer. It is the static trace-point creation mechanism.

```
#define INCL_TRACE
#include <os2.h>

PTCEREQUEST    pTraceCreateEntry;
APIRET         rc;

rc = TraceCreateEntry(pTraceCreateEntry);
```

TraceCreateEntry Parameter - pTraceCreateEntry

pTraceCreateEntry (PTCEREQUEST) - input
Pointer to the TraceCreateEntry parameter packet.

TraceCreateEntry Return Value - rc

rc (APIRET) - returns
TraceCreateEntry returns the following values:

- 1702 INVALID_LF_PACKET_REVISION_NUMBER
- 1703 INVALID_DATA_POINTER
- 1802 INVALID_TRACE_MAJOR_CODE
- 1803 INVALID_TRACE_MINOR_CODE
- 1806 INVALID_PACKET_SIZE

TraceCreateEntry - Parameters

pTraceCreateEntry (PTCEREQUEST) - input
Pointer to the TraceCreateEntry parameter packet.

rc (APIRET) - returns
TraceCreateEntry returns the following values:

- 1702 INVALID_LF_PACKET_REVISION_NUMBER
- 1703 INVALID_DATA_POINTER
- 1802 INVALID_TRACE_MAJOR_CODE
- 1803 INVALID_TRACE_MINOR_CODE
- 1806 INVALID_PACKET_SIZE

TraceCreateEntry - Remarks

Event trace records contain information that describes the occurrence of software events. They can be used both as service aids and in performance monitoring.

The library TRACE.LIB must be linked with object files that use TraceCreateEntry.

TraceCreateEntry - Example Code

The following example adds an event trace entry to the system trace buffer. For this example, the trace entry will contain the contents of two internal program variables.

```
#define INCL_DOSPROCESS

#include <stdio.h>                    /* C library for standard I/O        */
#include <stdlib.h>                   /* C library of standard routines    */
#include <string.h>                   /* C library for string operations   */
#include <os2.h>                      /* OS/2 Dos api calls                */
#include <trace.h>                    /* Trace public API data structures  */

#define HKWD_TEST 43
#define hkwd_test_entry 0001

struct {
  ULONG  var1;
  USHORT var2;
} trace_data;

TCEREQUEST trace_create_entry_packet;

VOID main(VOID)
{
  APIRET rc = NO_ERROR;

  /*************************************************************************/
  /* Set up the TraceCreateEntry parameter packet                          */
  /*************************************************************************/
  trace_create_entry_packet.packet_size            = sizeof(TCEREQUEST);
  trace_create_entry_packet.packet_revision_number = TRACE_RELEASE;
  trace_create_entry_packet.major_event_code       = HKWD_TEST;
  trace_create_entry_packet.minor_event_code       = hkwd_test_entry;
  trace_create_entry_packet.event_data_length      = sizeof(trace_data);
  trace_create_entry_packet.event_data             = (PVOID)&trace_data;

  /*************************************************************************/
  /* Place tracepoint data in the tracepoint data buffer                   */
  /*************************************************************************/
  trace_data.var1 = UINT_MAX;
  trace_data.var2 = 1;

  rc = TraceCreateEntry(&trace_create_entry_packet);
  if (rc |= NO_ERROR) {
    printf("TraceCreateEntry RC(%d)\n", rc);
  }
}
```

TraceCreateEntry - Topics

Select an item:

Problem Determination Data Types

This section describes the following data types that are used with the Problem Determination functions:

- CONFIGPARMS

- DISABLEPRODUCT
- DMIDATA
- DUMPDATAVAR
- DUMPUSERDATA
- EVENTKEY
- FFSTPARMS
- FILTERBLOCK
- HEADERBLOCK
- LCEFREQUEST
- LCENREQUEST
- LCFREQUEST
- LFEREQUEST
- LOENREQUEST
- LOFREQUEST
- LREREQUEST
- LWEREQUEST
- MODINFO
- MSGINSDATA
- MSGINSTXT
- PRODUCTDATA
- PRODUCTINFO
- SUBBLOCK
- TCEREQUEST

CONFIGPARMS

```
typedef struct _CONFIGPARMS {
  ULONG               packet_size;
  ULONG               packet_revision_number;
  ULONG               dump_file_wrap;
  UniChar            *dump_file_directory_name_length;
  PVOID               dump_file_directory_name;
  ULONG               no_of_probe_disabled_products;
  PDISABLEPRODUCT     pDisableProduct;
  ULONG               message_pop_up;
} CONFIGPARMS;

typedef CONFIGPARMS *PCONFIGPARMS;
```

CONFIGPARMS Field - packet_size

packet_size (ULONG)
Size, in bytes, of this packet.

CONFIGPARMS Field - packet_revision_number

packet_revision_number (ULONG)
A long integer value that indicates the revision level of the parameter packet. Use one of the following values defined in FFST.H:

UniCode parameter packet.	Use CONFIGPARMS_OS2_UNICODE to specify the UniCode parameter packet.
ASCII parameter packet.	Use CONFIGPARMS_OS2_ASCII to specify the ASCII parameter packet.

CONFIGPARMS Field - dump_file_wrap

dump_file_wrap (ULONG)
Indicates whether the dumps captured by FFST are to be wrapped or not.

If dump wrap is set to on then dumps will be wrapped when the specified maximum size is reached. If dump wrap is set to off when the specified maximum size is reached then FFST will not create any new dump files.

May be one of the following values:

0	Set dump wrap to off
1	Set dump wrap to on (system default)

CONFIGPARMS Field - dump_file_directory_name_length

dump_file_directory_name_length (ULONG)
Length, in bytes, of the dump directory name.

CONFIGPARMS Field - dump_file_directory_name

dump_file_directory_name (PVOID)
Pointer to the fully-qualified dump directory name.

CONFIGPARMS Field - no_of_probe_disabled_products

no_of_probe_disabled_products (ULONG)
Number of products whose FFSTProbes are disabled.

CONFIGPARMS Field - pDisableProduct

pDisableProduct (PDISABLEPRODUCT)

Pointer to the structure that identifies the product data of the disabled probe. This parameter can be ignored when *no_of_probe_disabled_products* is 0 or 'FFFFFFFF'.

CONFIGPARMS Field - message_pop_up

message_pop_up (ULONG)
Reserved parameter.

DISABLEPRODUCT

```
typedef struct _DISABLEPRODUCT {
  UniChar      *DMI_vendor_tag;
  UniChar      *DMI_tag;
  UniChar      *DMI_revision;
} DISABLEPRODUCT;

typedef DISABLEPRODUCT *PDISABLEPRODUCT;
```

DISABLEPRODUCT Field - DMI_vendor_tag

DMI_vendor_tag (UniChar *)
Pointer to the short product manufacturer name that was logged in the DMI database.

DISABLEPRODUCT Field - DMI_tag

DMI_tag (UniChar *)
Pointer to the short product name that was logged in the DMI database.

DISABLEPRODUCT Field - DMI_revision

DMI_revision (UniChar *)
Pointer to the product revision information that was logged in the DMI database.

DMIDATA

```
typedef struct _DMIDATA {
  ULONG         packet_size;
  ULONG         packet_revision_number;
  PVOID         DMI_product_ID;
  PVOID         DMI_modification_level;
  PVOID         DMI_fix_level;
  ULONG         template_filename_length;
  PVOID         template_filename;
} DMIDATA;

typedef DMIDATA *PDMIDATA;
```

Use appropriate *packet_revision_number* to indicate whether character data in the parameter packet is in UniCode or ASCII format.

DMIDATA Field - packet_size

packet_size (ULONG)
Length, in bytes, of this packet.

DMIDATA Field - packet_revision_number

packet_revision_number (ULONG)
A long integer value that indicates the revision level of the parameter packet. Use one of the following values defined in FFST.H:

UniCode parameter packet.	Use DMIDATA_UNICODE to specify the UniCode parameter packet.
ASCII parameter packet.	Use DMIDATA_ASCII to specify the ASCII parameter packet.

DMIDATA Field - DMI_product_ID

DMI_product_ID (PVOID)
Pointer to the product ID.

DMIDATA Field - DMI_modification_level

DMI_modification_level (PVOID)
Pointer to the product modification level.

DMIDATA Field - DMI_fix_level

DMI_fix_level (PVOID)
Pointer to the product fix level.

DMIDATA Field - template_filename_length

template_filename_length (ULONG)
Length of the template repository file name pointed to by *template_filename* .

DMIDATA Field - template_filename

template_filename (PVOID)
Pointer to the fully-qualified path of the template repository file.

DUMPDATAVAR

```
typedef struct _DUMPDATAVAR {
  ULONG     var_n_length;
  PVOID     var_n;
} DUMPDATAVAR;

typedef DUMPDATAVAR *PDUMPDATAVAR;
```

DUMPDATAVAR Field - var_n_length

var_n_length (ULONG)
Length, in bytes, of the data structure pointed to by *var_n* .

DUMPDATAVAR Field - var_n

var_n (PVOID)
Pointer to the data structure to be collected.

DUMPUSERDATA

```
typedef struct _DUMPUSERDATA {
  ULONG           no_of_variables;
  DUMPDATAVAR     DumpDataVar[MAX_USER_DUMPS];
} DUMPUSERDATA;

typedef DUMPUSERDATA *PDUMPUSERDATA;
```

DUMPUSERDATA Field - no_of_variables

no_of_variables (ULONG)
Number of data/structures to be collected by the probe. Maximum number of variables that can be collected is 30.

DUMPUSERDATA Field - DumpDataVar[MAX_USER_DUMPS]

DumpDataVar[MAX_USER_DUMPS] (DUMPDATAVAR)
Dump data variables repeated *no_of_variables* times.

EVENTKEY

Event key data structure.

```
typedef struct _EVENTKEY {
  ULONG     location;
  ULONG     entry_ID;
} EVENTKEY;

typedef EVENTKEY *PEVENTKEY;
```

EVENTKEY Field - location

location (ULONG) - output
An unsigned long word that contains a relative location within the log file. This value will point to the beginning of a log entry.

EVENTKEY Field - entry_ID

entry_ID (ULONG) - output
Entry ID of the record.

FFSTPARMS

```
typedef struct _FFSTPARMS {
  ULONG             packet_size;
  ULONG             packet_revision_number;
  PVOID             module_name;
  ULONG             probe_ID;
  ULONG             severity;
  ULONG             template_record_ID;
  PMSGINSDATA       pMsgInsData;
  ULONG             probe_flags;
  PDUMPUSERDATA     pDumpUserData;
  ULONG             log_user_data_length;
  PVOID             log_user_data;
} FFSTPARMS;

typedef FFSTPARMS *PFFSTPARMS;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

FFSTPARMS Field - packet_size

packet_size (ULONG)
Length, in bytes, of the packet.

FFSTPARMS Field - packet_revision_number

packet_revision_number (ULONG)
A long integer value that indicates the revision level of the parameter packet. Use one of the following values defined in FFST.H:

UniCode parameter packet.	Use FFSTPARMS_OS2_UNICODE to specify the UniCode parameter packet.
ASCII parameter packet.	Use FFSTPARMS_OS2_ASCII to specify the ASCII parameter packet.

FFSTPARMS Field - module_name

module_name (PVOID)
Address of the module name.

FFSTPARMS Field - probe_ID

probe_ID (ULONG)
Unique identifier of the detection point within the module_name.

FFSTPARMS Field - severity

severity (ULONG)
The severity of this error. Valid values are 1 (highest) through 6 (lowest):

1	SEVERITY1 *Critical:* Condition from which there is no recovery.
2	SEVERITY2 *Major:* Condition signifying that the loss of availability of a device or subproduct is imminent, or the performance of the device or subproduct has degraded to below an acceptable level.
3	SEVERITY3 *Minor:* Condition that was recovered from after a number of attempts or a nonservice-affecting fault has occurred that should be corrected before a more serious error occurs.
4	SEVERITY4 *Warning:* A potential error has been detected before any significant effects have been felt.
5	SEVERITY5 *Indeterminate:* Condition where it is not possible to determine the severity of the error.
6	SEVERITY6 *Information*

For severity 1, 2 and 3, FFST will request trace information, if system trace is 'ON'. For severity 4, 5 and 6, FFST will not request trace information, unless the end user specifically requests it through a probe control table entry.

FFSTPARMS Field - template_record_ID

template_record_ID (ULONG)
Identifier to specify the Log Entry Format Template that is be used by SYSLOG utility.

FFSTPARMS Field - pMsgInsData

pMsgInsData (PMSGINSDATA)
A pointer to message insert data. NULL indicates that there are no insert texts for the message.

FFSTPARMS Field - probe_flags

probe_flags (ULONG)
Indicates what type of system information is needed and where to keep the information that is collected. If PSTAT or Process Environment are requested, FFST will create a dump file and will keep this information in the dump file. The user can specify a combination of these flags. As an example, if it is desired to capture both PSTAT and Process Environment information, then the value specified will be 3.

If the user sets any of the probe flags listed below, FFST will create an FFST dump file where the information will be stored. The exceptions are Process Dump (PROCESS_DUMP_FLAG) and System Trace (CAPTURE_TRACE). These processes will have their own files that FFST will maintain. Trace information will be automatically generated for severity 1, 2, and 3 probes. For severity 4, 5 and 6 probes, trace will not be collected.

PSTAT_FLAG (0x01) Capture PSTAT information.
PROC_ENV_FLAG (0x02) Capture process environment information.

FFSTPARMS Field - pDumpUserData

pDumpUserData (PDUMPUSERDATA)
Pointer to user_data containing various storage areas. Maximum storage areas that can be specified are 30. NULL indicates that there are no user data items for this probe call. The items specified will be stored in a dump file created by FFST. Each individual data area can have a maximum size of 32K.

FFSTPARMS Field - log_user_data_length

log_user_data_length (ULONG)
The length in bytes of *log_user_data* . A length of 0 means there is no user data.

FFSTPARMS Field - log_user_data

log_user_data (PVOID)
Pointer to data to be placed in the Error Log File. The data can be formatted through the template for display. If no user data is desired, then the pointer must be null. This data will be stored in the Error Log File as part of the log entry.

FILTERBLOCK

Filter block structure.

```
typedef struct _FILTERBLOCK {
  ULONG            packet_size;
  ULONG            packet_revision_number;
  PHEADERBLOCK     header_block;
} FILTERBLOCK;

typedef FILTERBLOCK *PFILTERBLOCK;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

FILTERBLOCK Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

FILTERBLOCK Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

FILTERBLOCK Field - header_block

header_block (PHEADERBLOCK) - input
Pointer to the first header block.

HEADERBLOCK

Header block for an event notification filter structure.

```
typedef struct _HEADERBLOCK {
  PSUBBLOCK                pSubblock;
  struct _HEADERBLOCK      *pNextBlock;
} HEADERBLOCK;

typedef HEADERBLOCK *PHEADERBLOCK;
```

HEADERBLOCK Field - pSubblock

pSubblock (PSUBBLOCK) - input
Pointer to the first sub-block.

HEADERBLOCK Field - pNextBlock

pNextBlock (struct _HEADERBLOCK *) - input
Pointer to the next header block in the chain. A NULL indicates the end of the chain.

LCEFREQUEST

LogChangeEventFilter parameter packet.

```
typedef struct _LCEFREQUEST {
  ULONG            packet_size;
  ULONG            packet_revision_number;
  ULONG            purge_flags;
  ULONG            LogNotify;
  PFILTERBLOCK     pFilter;
} LCEFREQUEST;

typedef LCEFREQUEST *PLCEFREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LCEFREQUEST Field - packet_size

packet_size (ULONG) - input

The size, in bytes, of this packet.

LCEFREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LCEFREQUEST Field - purge_flags

purge_flags (ULONG) - input
Indicates whether the contents of the event notification mechanism should be purged when the event filter is changed.

Value	Defintion
0	Do not purge the existing contents before the filter change takes effect The KEEP_EVENT_NOTIFICATION value defined in LFDEF.H can also be used for this parameter.
1	Purge the existing contents before the filter change takes effect The PURGE_EVENT_NOTIFICATION value defined in LFDEF.H can also be used for this parameter.

LCEFREQUEST Field - LogNotify

LogNotify (ULONG) - input
ID of event notification mechanism for which filtering is to be changed.

LCEFREQUEST Field - pFilter

pFilter (PFILTERBLOCK) - input
A pointer to the new event notification filter data structure. A NULL pointer indicates that no filter is specified.

The valid set of error log entry attribute IDs and comparison operators is the same as was specified for the *LogOpenEventNotification* API pFilter parameter.

LCENREQUEST

LogCloseEventNotification parameter packet.

```
typedef struct _LCENREQUEST {
  ULONG     packet_size;
  ULONG     packet_revision_number;
  ULONG     LogNotify;
} LCENREQUEST;

typedef LCENREQUEST *PLCENREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LCENREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LCENREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LCENREQUEST Field - LogNotify

LogNotify (ULONG) - input
ID of the event notification to be closed.

LCFREQUEST

LogCloseFile parameter packet.

```
typedef struct _LCFREQUEST {
  ULONG            packet_size;
  ULONG            packet_revision_number;
  ULONG            log_file_ID;
} LCFREQUEST;
```

```
typedef LCFREQUEST *PLCFREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LCFREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LCFREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LCFREQUEST Field - log_file_ID

log_file_ID (ULONG) - input
ID of the file to be closed.

LFEREQUEST

LogFormatEntry parameter packet.

```
typedef struct _LFEREQUEST {
  ULONG         packet_size;
  ULONG         packet_revision_number;
  PVOID         log_entry_buffer;
  PVOID         locale_object;
  PULONG        number_of_detail_records;
  ULONG         flags;
  PULONG        string_buffer_length;
  PVOID         string_buffer;
} LFEREQUEST;

typedef LFEREQUEST *PLFEREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LFEREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LFEREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LFEREQUEST Field - log_entry_buffer

log_entry_buffer (PVOID) - input
Pointer to the Log Entry record to be formatted.

LFEREQUEST Field - locale_object

locale_object (PVOID) - input
A pointer to the locale object. Not required when using ASCII data.

LFEREQUEST Field - number_of_detail_records

number_of_detail_records (PULONG) - output
The number of records that were returned in the buffer.

LFEREQUEST Field - flags

flags (ULONG) - input/output
A 4-Byte flags word that contains the following bit flags:

Bit	Value	Description
1	Input	(ERR_FORMAT_DETAIL_DATA 0x00000001) Indicates that the user data portion of the error record should be formatted. If not set, the user data will be returned using the hex format.
2	Input	(FORMAT_ONLY_ERROR_DESCRIPTION 0x00000010) Only the error description portion of the record will be formatted.
4	Output	(TEMPLATE_NOT_FOUND 0x00000100) Formatting of this record could not be accomplished because the specified template record could not be found.
8	Output	(ERROR_DESCRIPTION_FILE_NOT_FOUND 0x00001000) The Error description file was not found.
16	Output	(CAUSE_MSG_FILE_NOT_FOUND 0x00010000) The Cause message file was not found.
32	Output	(ACTION_MSG_FILE_NOT_FOUND 0x00100000) The Actions message file was not found.
64	Output	(DETAIL_DATA_FILE_WAS_NOT_FOUND 0x01000000) The Detail Data file was not found.

All other values are reserved for future use.

LFEREQUEST Field - string_buffer_length

string_buffer_length (PULONG) - input/output
On input, the length of the caller's formatted string buffer.

On output, the total number of bytes of formatting strings that are placed within the formatted string buffer. If the callers's buffer is too small to fit all the formatted strings, the this will be set to the required size and no data will be placed in the buffer.

LFEREQUEST Field - string_buffer

string_buffer (PVOID) - input
Pointer to the output buffer that will be filled with a set of formatted strings.

The packet_revision_number parameter determines if character data is returned as ASCII or UniCode character data.

LOENREQUEST

LogOpenEventNotification parameter packet.

```
typedef struct _LOENREQUEST {
  ULONG            packet_size;
  ULONG            packet_revision_number;
  ULONG            log_file_ID;
  ULONG            read_flags;
  PULONG           pLogNotify;
  PFILTERBLOCK     pFilter;
} LOENREQUEST;
```

```
typedef LOENREQUEST *PLOENREQUEST;
```

Use appropriate *packet_revision_number* to indicate whether character data in the parameter packet is in UniCode or ASCII format.

LOENREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LOENREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LOENREQUEST Field - log_file_ID

log_file_ID (ULONG) - input
An integer ID that describes which log file the event notification is to be associated with. This is restricted to the current log file which has an ID of 1.

LOENREQUEST Field - read_flags

read_flags (ULONG) - input
Flag values used to control the notification.

0	Return the record
1	Do not return the record.

The RETURN_NO_DATA value defined in LFDEF.H can also be used for this parameter.

LOENREQUEST Field - pLogNotify

pLogNotify (PULONG) - output
A pointer to a variable that will receive the ID of event notification mechanism.

This ID will be used on future LogWaitEvent calls to specify which event notification mechanism is being used.

LOENREQUEST Field - pFilter

pFilter (PFILTERBLOCK) - input
Pointer to the event filter data structure. A NULL indicates that no initial filter is specified.

The following attribute IDs are defined for error log entries. They identify the fields of an error log entry that can be used within an event notification filter:

- LOG_ERROR_DATE
- LOG_ERROR_TIME
- LOG_ERROR_ENTRY_ID
- LOG_ERROR_RECORD_TYPE
- LOG_ERROR_SEVERITY
- LOG_ERROR_PROCESS_PATHNAME
- LOG_ERROR_SOURCE_MODULE_NAME
- LOG_ERROR_PROBE_ID
- LOG_ERROR_DMI_VENDOR_TAG
- LOG_ERROR_DMI_TAG
- LOG_ERROR_DMI_REVISION
- LOG_ERROR_MACHINE_TYPE
- LOG_ERROR_SERIAL_NUMBER
- LOG_ERROR_USER_DATA

The following comparison operator IDs are defined for the error logging service. They identify comparison operators that can be used within an event notification filter:

- LOG_ERROR_EQUAL
- LOG_ERROR_NOT_EQUAL
- LOG_ERROR_GREATER_THAN
- LOG_ERROR_GREATER_THAN_OR_EQUAL
- LOG_ERROR_LESS_THAN
- LOG_ERROR_LESS_THAN_OR_EQUAL
- LOG_ERROR_SUBSTRING_MATCH

LOFREQUEST

LogOpenFile parameter packet.

```
typedef struct _LOFREQUEST {
  ULONG         packet_size;
  ULONG         packet_revision_number;
  PULONG        log_file_ID;
  PULONG        filename_length;
  PVOID         filename;
} LOFREQUEST;

typedef LOFREQUEST *PLOFREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LOFREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LOFREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LOFREQUEST Field - log_file_ID

log_file_ID (PULONG) - input/output
A pointer to an area that contains an integer ID of the file to be opened.

On input, specifies the number of the file that you wish to open. There can be various combinations of log_file_ID and filename requests. The following table shows what will happen in each case.

Contents of log_file_ID	Contents of file_name	What happens
A number was entered	A name was entered	The file_name is ignored If log_file_ID exists, the file is opened and a new log_file_ID number is assigned. If log_file_ID does not exist, a failing return code is returned.
A number was entered	NULL	If log_file_ID exists, the file is opened and a new log_file_ID number is assigned. If log_file_ID does not exist, a failing return code is returned.
NULL	A name was entered	If the file exists, the file is opened and a new log_file_ID number is assigned. If the file does not exist, a failing return code is returned.

On output, contains the number of the file that was opened.

LOFREQUEST Field - filename_length

filename_length (PULONG) - input/output
Length of filename.

On input, specifies the length of the filename that you are opening. A length of 0 specifies that no filename was specified.

On output, the length of the filename that corresponds to the log_file_ID that was opened.

LOFREQUEST Field - filename

filename (PVOID) - input/output
Pointer to the filename.

On input, specifies the fielname that you are opening. This may be NULL if you specify a log_file_ID number.

On output, the filename that corresponds to the log_file_ID that was opened.

The packet_revision_number parameter defines if this pointer points to ASCII or UniCode character data.

LREREQUEST

LogReadEntry parameter packet.

```
typedef struct _LREREQUEST {
  ULONG            packet_size;
  ULONG            packet_revision_number;
  ULONG            log_file_ID;
  ULONG            flags;
  PEVENTKEY        pEventKey;
  PFILTERBLOCK     pFilter;
  PULONG           pLogEntryBufferLength;
  PVOID            pLogEntryBuffer;
} LREREQUEST;

typedef LREREQUEST *PLREREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LREREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LREREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LREREQUEST Field - log_file_ID

log_file_ID (ULONG) - input
ID of the log file to be used by the logging service. This would normally be the log_file_ID returned from the LogOpenFile API.

LREREQUEST Field - flags

flags (ULONG) - input
Indicates the relative position to start the read and the amount of data to return.

Bit Value	Definition
No bits set	Start with the entry prior to the one specified in the event key structure.
1	Start with the entry prior to the one specified in the event key structure.
2	Start at the most recent entry.
4	Start at the oldest (last) record on file. This request will not be limited by in formation in the event key or filter structures. **Note:** The multiple record flag is NOT valid for this request; only the oldest (last) record on file will be returned to the caller.
8	Start at the entry identified by the EventKey.

Note: the previous bit settings are mutually exclusive. If multiples are set ON, the most recent entry will be returned.

16	Multiple Read; Return as many complete records as possible (limited by buffer size or End-of-file condition). Upon completion, the event key structure contains information pertaining to the last record returned to the caller. **Note:** This flag will be ignored if specified along with an oldest record request.

LREREQUEST Field - pEventKey

pEventKey (PEVENTKEY) - input/output
Pointer to the event key data structure.

When used as an input parameter to this function, you can specify which entry is to be read. For example, when an event key is returned from LogWaitEvent API, the event key can be used to read a particular entry from the system error log.

LREREQUEST Field - pFilter

pFilter (PFILTERBLOCK) - input
Pointer to the event filter to use for the search.

The valid set of error log entry attribute IDs and comparison operators is the same as was specified for the *LogOpenEventNotification* API pFilter parameter.

LREREQUEST Field - pLogEntryBufferLength

pLogEntryBufferLength (PULONG) - input/output
On input, length of caller's buffer.

On output, the total number of bytes placed in the provided buffer. If the caller's buffer is too small, it will be set to the required size.

If the packet_revision_number parameter specifies ASCII character data, the data returned will be in ASCII format.

If the packet_revision_number parameter specifies UniCode character data, the data returned will be in UniCode.

LREREQUEST Field - pLogEntryBuffer

pLogEntryBuffer (PVOID) - input
Pointer to the buffer containing the log record.

The packet_revision_number parameter defines if pointers point to ASCII or UniCode character data.

LWEREQUEST

LogWaitEvent parameter packet.

```
typedef struct _LWEREQUEST {
  ULONG         packet_size;
  ULONG         packet_revision_number;
  ULONG         LogNotify;
  PEVENTKEY     pEventKey;
  PULONG        log_entry_buffer_length;
  PVOID         pLogEntryBuffer;
  ULONG         timeout;
  ULONG         queue_flags;
  ULONG         pathname_length;
  PVOID         pathname;
} LWEREQUEST;

typedef LWEREQUEST *PLWEREQUEST;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

LWEREQUEST Field - packet_size

packet_size (ULONG) - input
The size, in bytes, of this packet.

LWEREQUEST Field - packet_revision_number

packet_revision_number (ULONG) - input
A long integer value that indicates the revision level of the parameter packet.

UniCode parameter packet. LF_UNI_API defined in LFDEF.H can also be used to specify the UniCode parameter packet for the revision level.
ASCII parameter packet. LF_ASCII_API defined in LFDEF.H can also be used to specify the ASCII parameter packet for the revision level.

LWEREQUEST Field - LogNotify

LogNotify (ULONG) - input
Log ID returned from event notification.

LWEREQUEST Field - pEventKey

pEventKey (PEVENTKEY) - input
Pointer to the event key data structure.

This event key data structure is returned by the logging service.

This data structure can subsequently be passed to LogReadEntry so that the event consumer can read the log file entry that is associated with this event notification.

LWEREQUEST Field - log_entry_buffer_length

log_entry_buffer_length (PULONG) - input/output
The size, in bytes, of the provided buffer pointed to by **pLogEntryBuffer**.

On input, this is the length of the caller's provided buffer.

On output, this is the total number of bytes placed in the provided buffer. If the caller's buffer is too small, then this will be set to the required size and no data will be written to the buffer.

LWEREQUEST Field - pLogEntryBuffer

pLogEntryBuffer (PVOID) - output
Pointer to the buffer that is to receive the selected error log record.

Note that the data returned in the buffer can be either an ASCII or UniCode data. The type of character string is determined by the packet_revision_number parameter.

LWEREQUEST Field - timeout

timeout (ULONG) - input
Milliseconds to wait before timing out. A value of 0 indicates to wait indefinitely.

LWEREQUEST Field - queue_flags

queue_flags (ULONG) - input
Flags that indicate how the queue will be handled if the buffer provided by the user is not large enough to hold the record.

May be one of the following valued:

0	Do not remove record from queue.
1	Remove record from queue. (If the record is still required, it can be retrieved from the log file by using the EVENTKEY structure and LogReadEntry.)

The DEQ_ON_ERROR value defined in LFDEF.H can also be used for this parameter.

LWEREQUEST Field - pathname_length

pathname_length (ULONG) - output
Length of the pathname.

This is the total number of bytes placed in *pathname*. If the caller's *pathname* length is too small, then this will be set to the required length and no data will be written to *pathname*.

LWEREQUEST Field - pathname

pathname (PVOID) - output
Pointer to the pathname where the error log record being reported in LogEntryBuffer resides.

The packet_revision_number parameter defines if pointers point to ASCII or UniCode character data.

MODINFO

```
typedef struct _MODINFO {
  ULONG        packet_size;
  ULONG        packet_revision_number;
  PUNICHAR     subproduct_name;
  PUNICHAR     module_name;
} MODINFO;

typedef MODINFO *PMODINFO;
```

MODINFO Field - packet_size

packet_size (ULONG)
Length, in bytes, of this packet.

MODINFO Field - packet_revision_number

packet_revision_number (ULONG)
Revision number for this packet.

MODINFO_REVISION_NUMBER_1 1

MODINFO Field - subproduct_name

subproduct_name (PUNICHAR)
Pointer to the subproduct name.

MODINFO Field - module_name

module_name (PUNICHAR)
Pointer to the module identifier name. Identifies the originator of the probe.

MSGINSDATA

```
typedef struct _MSGINSDATA {
  ULONG         no_inserts;
  MSGINSTXT     MsgInsTxt[MAX_MSG_INSERTS];
} MSGINSDATA;

typedef MSGINSDATA *PMSGINSDATA;
```

MSGINSDATA Field - no_inserts

no_inserts (ULONG)
Number of insert strings provided. Maximum allowed is 9.

MSGINSDATA Field - MsgInsTxt[MAX_MSG_INSERTS]

MsgInsTxt[MAX_MSG_INSERTS] (MSGINSTXT)
Message insert text details repeated *no_inserts* times.

MSGINSTXT

```
typedef struct _MSGINSTXT {
  ULONG         insert_number;
  PVOID         insert_text;
} MSGINSTXT;

typedef MSGINSTXT *PMSGINSTXT;

The packet_revision_number parameter defines if pointers point
to ASCII or UniCode character data.
```

MSGINSTXT Field - insert_number

insert_number (ULONG)
The insert number. This must match the %n in the message.

MSGINSTXT Field - insert_text

insert_text (UniChar *)
Address of the message text.

PRODUCTDATA

```
typedef struct _PRODUCTDATA {
  ULONG         packet_size;
  ULONG         packet_revision_number;
  PVOID         DMI_vendor_tag;
  PVOID         DMI_tag;
  PVOID         DMI_revision;
} PRODUCTDATA;

typedef PRODUCTDATA *PPRODUCTDATA;

Use appropriate packet_revision_number to indicate whether character
data in the parameter packet is in UniCode or ASCII format.
```

PRODUCTDATA Field - packet_size

packet_size (ULONG)
Length, in bytes, of this packet.

PRODUCTDATA Field - packet_revision_number

packet_revision_number (ULONG)
A long integer value that indicates the revision level of the parameter packet. Use one of the following values defined in FFST.H:

UniCode parameter packet.	Use PRODUCTDATA_UNICODE to specify the UniCode parameter packet.
ASCII parameter packet.	Use PRODUCTDATA_ASCII to specify the ASCII parameter packet.

PRODUCTDATA Field - DMI_vendor_tag

DMI_vendor_tag (PVOID)
Pointer to the name of the product manufacturer.

PRODUCTDATA Field - DMI_tag

DMI_tag (PVOID)
Pointer to the product name that was logged in the DMI database.

PRODUCTDATA Field - DMI_revision

DMI_revision (PVOID)
Pointer to the product version number.

PRODUCTINFO

```
typedef struct _PRODUCTINFO {
  PPRODUCTDATA     pProductData;
  PDMIDATA         pDMIData;
} PRODUCTINFO;

typedef PRODUCTINFO *PPRODUCTINFO;
```

PRODUCTINFO Field - pProductData

pProductData (PPRODUCTDATA)
Pointer to a structure that identifies the module in which the probe is inserted.

PRODUCTINFO Field - pDMIData

pDMIData (PDMIDATA)
Pointer to a structure that specifies the product information if DMI is not used. A NULL value indicates that the DMI information will be retrieved from the DMI Repository (recommended approach).

SUBBLOCK

Selection criteria sub-block structure.

```
typedef struct _SUBBLOCK {
  ULONG                 entry_attribute_ID;      /*  The identifier of a field within a log entry. */
  ULONG                 comparison_operator_ID;  /*  The identifier of a comparison operator. */
  ULONG                 comparison_data_length;  /*  The length of the comparison data. */
  PVOID                 comparison_data_ptr;
  struct _SUBBLOCK      *next_subblock;
} SUBBLOCK;

typedef SUBBLOCK *PSUBBLOCK;
```

Note that the **comparison_data_ptr** can point to either ASCII or UniCode character data. The type of character data is determined by the *packet_revision_number* parameter of FILTERBLOCK.

SUBBLOCK Field - entry_attribute_ID

entry_attribute_ID (ULONG) - input
The identifier of a field within a log entry.

When character data that is pointed to, the data can be either ASCII or UniCode data. Use the appropriate packet_revision_number to specify the data type.

The field within an entry that will be compared against the target data value that is pointed to by *comparison_data_ptr* .

May be one of the following values for Error Logging:

LOG_ERROR_DATE	Date type
LOG_ERROR_TIME	Time type
LOG_ERROR_ENTRY_ID	Unsigned long integer type
LOG_ERROR_RECORD_TYPE	String type
LOG_ERROR_SEVERITY	Unsigned long integer type
LOG_ERROR_PROCESS_PATHNAME	String type
LOG_ERROR_SOURCE_MODULE_NAME	String type
LOG_ERROR_PROBE_ID	Unsigned long integer type
LOG_ERROR_DMI_VENDOR_TAG	String type
LOG_ERROR_DMI_TAG	String type
LOG_ERROR_DMI_REVISION	String type
LOG_ERROR_MACHINE_TYPE	String type
LOG_ERROR_SERIAL_NUMBER	String type
LOG_ERROR_USER_DATA	String type

SUBBLOCK Field - comparison_operator_ID

comparison_operator_ID (ULONG) - input

The identifier of a comparison operator.

This comparison operator must be valid for the type of log entry data item that was specified by *entry_attribute_ID* .

May be one of the following values:

LOG_ERROR_EQUAL	Date, time, string, and unsigned long integer types
LOG_ERROR_NOT_EQUAL	Date, time, string, and unsigned long integer types
LOG_ERROR_GREATER_THAN	Date, time, string, and unsigned long integer types
LOG_ERROR_GREATER_THAN_OR_EQUAL	Date, time, string, and unsigned long integer types
LOG_ERROR_LESS_THAN	Date, time, string, and unsigned long integer types
LOG_ERROR_LESS_THAN_OR_EQUAL	Date, time, string, and unsigned long integer types
LOG_SUBSTRING_MATCH	String type only

SUBBLOCK Field - comparison_data_length

comparison_data_length (ULONG) - input
The length of the comparison data.

SUBBLOCK Field - comparison_data_ptr

comparison_data_ptr (PVOID) - input
Pointer to a data item that will be compared against the specified log entry attribute.

Note: The data item is expected to be in the proper format. An example is the date and time attributes, which require data in the format that is maintained in the Log File.

SUBBLOCK Field - next_subblock

next_subblock (struct _SUBBLOCK *) - input
Pointer to the next sub-block in the chain. A NULL indicate the end of the chain.

TCEREQUEST

Structure for TraceCreateEntry.

```
typedef struct _TCEREQUEST {
  ULONG     packet_size;
```

```
  ULONG     packet_revision_number;
  ULONG     major_event_code;
  ULONG     minor_event_code;
  ULONG     event_data_length;
  PVOID     event_data;
} TCEREQUEST;

typedef TCEREQUEST *PTCEREQUEST;
```

TCEREQUEST Field - packet_size

packet_size (ULONG)
The size, in bytes, of this packet.

TCEREQUEST Field - packet_revision_number

packet_revision_number (ULONG)
A long integer value that indicates the revision level of the parameter packet.

2	Trace release number. TRACE_RELEASE defined in TRACE.H can also be used to specify the release number.

TCEREQUEST Field - major_event_code

major_event_code (ULONG)
The major event code of the event to be logged (must be between 0 and 255).

TCEREQUEST Field - minor_event_code

minor_event_code (ULONG)
The minor event code of the event to be logged (must be between 1 and 65535).

TCEREQUEST Field - event_data_length

event_data_length (ULONG)
The length of the data, in bytes, that is contained within the caller's event data buffer.

TCEREQUEST Field - event_data

event_data (PVOID)
The pointer to a buffer that contains the event data to be logged. The format of the data is specific to each tracepoint.
*

Notices

September 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:
FFST
First Failure Support Technology
IBM
OS/2
OS/2 Warp
OS/2 Warp Connect

The following terms are trademarks of other companies:

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.
